

WYDZIAŁ PODSTAWOWYCH PROBLEMÓW TECHNIKI  
POLITECHNIKA WROCŁAWSKA

# STEROWANIE MODELEM KOMPUTEROWYM Z WYKORZYSTANIEM SIECI NEURONOWEJ

BARTOSZ RZEPKOWSKI

NR INDEKSU: 208732

Praca inżynierska napisana  
pod kierunkiem  
Dr inż. Małgorzaty Sulkowskiej



Politechnika  
Wrocławska

WROCŁAW 2016

# Spis treści

<b>Wstęp</b>	<b>iii</b>
<b>1 Opis mechanizmu działania oraz typów sztucznych sieci neuronowych</b>	<b>1</b>
1.1 Struktura rzeczywistej komórki nerwowej . . . . .	1
1.2 Struktura neuronu wykorzystywanego w sieciach neuronowych . . . . .	2
1.3 Schemat prostej sieci neuronowej . . . . .	4
1.4 Proces uczenia sieci . . . . .	5
1.4.1 Podstawy uczenia sieci neuronowej . . . . .	5
1.4.2 Propagacja wsteczna . . . . .	6
1.4.3 Uczenie z nauczycielem . . . . .	7
1.4.4 Samouczenie sieci neuronowej . . . . .	7
1.4.5 Sieci samoorganizujące się . . . . .	7
1.4.6 Uczenie się ze wzmocnieniem . . . . .	8
<b>2 Przedstawienie narzędzi wybranych do realizacji projektu</b>	<b>11</b>
<b>3 Opis implementacji modelu komputerowego oraz sieci neuronowej</b>	<b>13</b>
3.1 Schemat modelu komputerowego . . . . .	13
3.2 Zarys działania programu . . . . .	14
3.2.1 Sterowanie modelem . . . . .	14
3.2.2 Proces uczenia się sieci neuronowej . . . . .	16
3.3 Kod źródłowy najważniejszych części projektu . . . . .	17
3.3.1 OnCollision . . . . .	17
3.3.2 MoveJoint . . . . .	18
3.3.3 Schemat komunikacji między ModelController oraz NeuralNetwork . . . . .	20
3.3.4 ModelController . . . . .	21
3.3.5 NeuralNetwork . . . . .	25
<b>4 Wyniki przeprowadzonych testów oraz płynące z nich wnioski</b>	<b>31</b>
4.1 Wyniki testów dla sieci z jedną warstwą ukrytą . . . . .	32
4.2 Wyniki testów dla sieci z dwoma warstwami ukrytymi . . . . .	33
4.3 Wnioski . . . . .	34
<b>Bibliografia</b>	<b>36</b>



# Wstęp

Celem niniejszej pracy jest stworzenie komputerowego modelu 3D sterowanego za pomocą sieci neuronowej w środowisku zbliżonym do rzeczywistego.

Motywacją do podjęcia się powyższego zadania była chęć stworzenia sieci neuronowej, która wygenerowałaby niejawną algorytm sterujący modelem oraz przetestowania jego zachowania w zależności od zmieniającej się struktury sieci.

Praca składa się z czterech rozdziałów. W pierwszym z nich opisano mechanizm działania oraz typy sieci neuronowych. W drugim przedstawiono wybrane narzędzia do realizacji projektu. W trzecim opisano implementację modelu komputerowego oraz sieci neuronowej. W czwartym rozdziale przedstawiono wyniki przeprowadzonych testów oraz płynące z nich wnioski.



# Opis mechanizmu działania oraz typów sztucznych sieci neuronowych

Sieć neuronowa to model matematyczny wzorowany na rzeczywistej strukturze mózgu. Jej zadaniem jest przetwarzanie sygnałów za pomocą wielu połączonych ze sobą elementów zwanych neuronami. Kluczowa dla powstania koncepcji sieci neuronowej była praca Warrena McCullocha oraz Waltera Pittsa. Starali się oni odkryć, w jaki sposób działają rzeczywiste neurony. W tym celu w 1943 roku zbudowali prosty obwód elektryczny odwzorowujący pracę mózgu.

W 1949 roku Donald Hebb w swojej książce "The Organization of Behavior" zauważył, że połączenia między komórkami nerwowymi są wzmacniane za każdym razem, gdy zostaną użyte. Obserwacja ta zaowocowała późniejszym powstaniem algorytmów umożliwiających uczenie się sieci neuronowych.

Przełomowym etapem w rozwoju tej dziedziny nauki było stworzenie przez Bernarda Widrowa i Marciana Hoffa w 1959 roku sieci ADALINE oraz MADALINE (z ang. Multiple ADaptive LINEar Elements). MADALINE miała za zadanie usuwanie szumów podczas połączeń telefonicznych. Mimo upływu wielu lat sieć ta wciąż wykorzystywana jest w przemyśle.

Sieci neuronowe doskonale nadają się do przewidywania danych wyjściowych na podstawie danych wejściowych, klasyfikacji oraz poszukiwania związków między danymi. Dzięki tym właściwościom znajdują zastosowanie w wielu obszarach, takich jak rozpoznawanie mowy, pisma i obrazów, prognozy giełdowe, diagnostyka medyczna czy sterowanie maszynami.

## 1.1 Struktura rzeczywistej komórki nerwowej

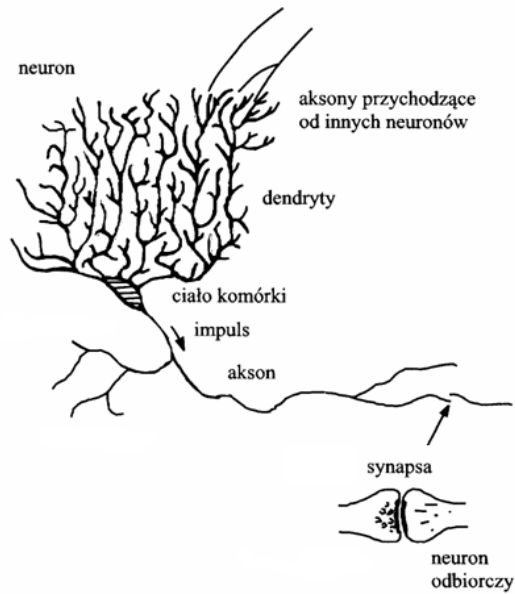
Głównymi elementami składającymi się na budowę neuronu są dendryty, ciało komórki nerwowej oraz akson. Dendryty pozwalają na odbieranie przez neuron sygnałów pochodzących z innych komórek nerwowych. Następnie impuls przechodzi przez ciało komórki oraz akson, który łączy się z dendrytami innych neuronów. W synapsie (miejscu połączenia się aksonu komórki wysyłającej sygnał z dendrytem neuronu odbiorczego) dochodzi do przekazania informacji za pomocą tzw. neuroprzekazników. Są to substancje chemiczne wydzielane przez akson, które odbierane są za pomocą specjalnie wyspecjalizowanych receptorów umieszczonych w dendrycie.

Proces nauki komórki nerwowej można przedstawić jako odpowiednie wzmacnianie lub osłabianie siły działania neuroprzekazników w miejscu połączenia neuronów.

Uproszczona struktura rzeczywistej komórki nerwowej przedstawiona jest na poniższej ilustracji<sup>1</sup>.

---

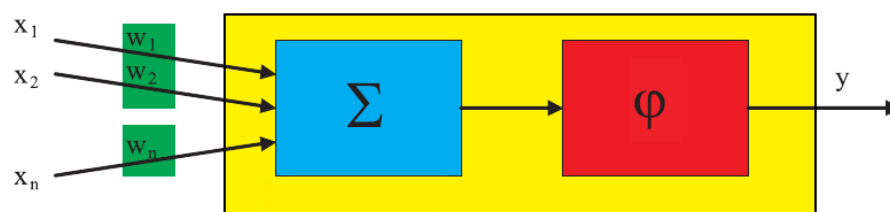
<sup>1</sup>Źródło: R. Tadeusiewicz, T. Gąciarz, B. Borowik, B. Leper - "Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#"



Rysunek 1.1: Uproszczona struktura rzeczywistego neuronu.

## 1.2 Struktura neuronu wykorzystywanego w sieciach neuronowych

Klasyczny model neuronu wykorzystywany przy tworzeniu sieci neuronowych jest niezwykle uproszczony w porównaniu z jego rzeczywistym odpowiednikiem. Jest to spowodowane niezwykle skomplikowaną strukturą realnej komórki nerwowej, której dokładne odwzorowanie jest utrudnione z powodów dzisiejszych ograniczeń sprzętowych. Ponadto, nie mamy jeszcze kompletnej wiedzy o tym, jak w mózgu przebiegają procesy myślowe. Poniższy rysunek przedstawia schemat sztucznego neuronu<sup>2</sup>.



Rysunek 1.2: Schemat neuronu wykorzystywanego w sieciach neuronowych.

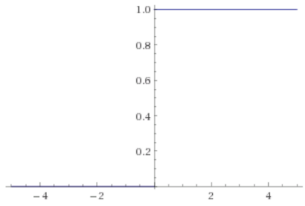
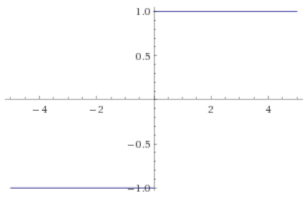
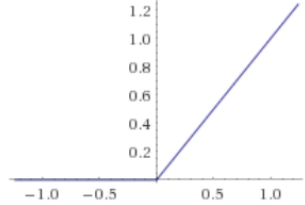
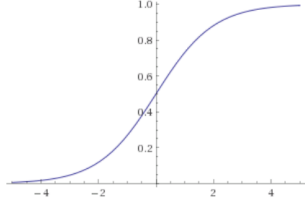
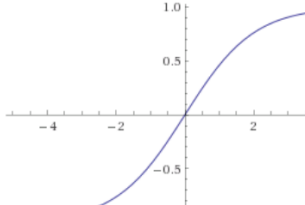
Sygnał wejściowy docierający do neuronu posiadającego  $n$  wejść (odpowiadających dendrytom rzeczywistej komórki nerwowej) możemy zapisać jako wektor  $\vec{x} = (x_1, x_2, x_3, \dots, x_n)$ . Na sygnał ten mogą się składać dane pochodzące bezpośrednio ze środowiska lub od innych neuronów. Do każdego wejścia przypisana jest waga (wzorowana na neuroprzekaźnikach występujących w synapsie). Wektor wszystkich wag możemy zapisać w postaci  $\vec{w} = (w_1, w_2, w_3, \dots, w_n)$ , a ich początkowe wartości są generowane w sposób losowy.

Na sygnał docierający do neuronu nakładana jest funkcja wejściowa postaci:

<sup>2</sup>Źródło: R. Tadeusiewicz, T. Gąciarz, B. Borowik, B. Leper - "Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#"

$$f_{\vec{w}}(\vec{x}) = \sum_{i=1}^n w_i * x_i.$$

Neuron, którego ostateczną odpowiedzią (ozn. jako  $y$ ) jest wynik powyższej funkcji, nazywamy neuro-nem liniowym. Jednak wynik ten może zostać również przekazany do następnej funkcji, nazywanej funkcją aktywacji. Jej przykłady zostały przedstawione w poniższej tabeli.

Funkcja aktywacji	Wzór matematyczny	Wykres
Funkcja progowa unipolarna	$\varphi(x) = \begin{cases} 0 & \text{dla } x < a \\ 1 & \text{dla } x \geq a \end{cases}$	
Funkcja progowa bipolarna	$\varphi(x) = \begin{cases} -1 & \text{dla } x < a \\ 1 & \text{dla } x \geq a \end{cases}$	
Funkcja ReLU (z ang. Rectified Linear Unit)	$\varphi(x) = \max\{0, x\}$	
Funkcja sigmoidalna unipolarna	$\varphi(x) = \frac{1}{1 + e^{-\beta x}}$	
Funkcja sigmoidalna bipolarna	$\varphi(x) = \frac{2}{1 + e^{-\beta x}} - 1$	





Funkcja Softmax	$\varphi(x) = \ln(1 + e^x)$	
-----------------	-----------------------------	--

Tablica 1.1: Przykłady funkcji aktywacji.

Funkcje progowe unipolarna oraz bipolarna mają za zadanie symulowanie dwóch stanów neuronu - aktywnego i nieaktywnego.

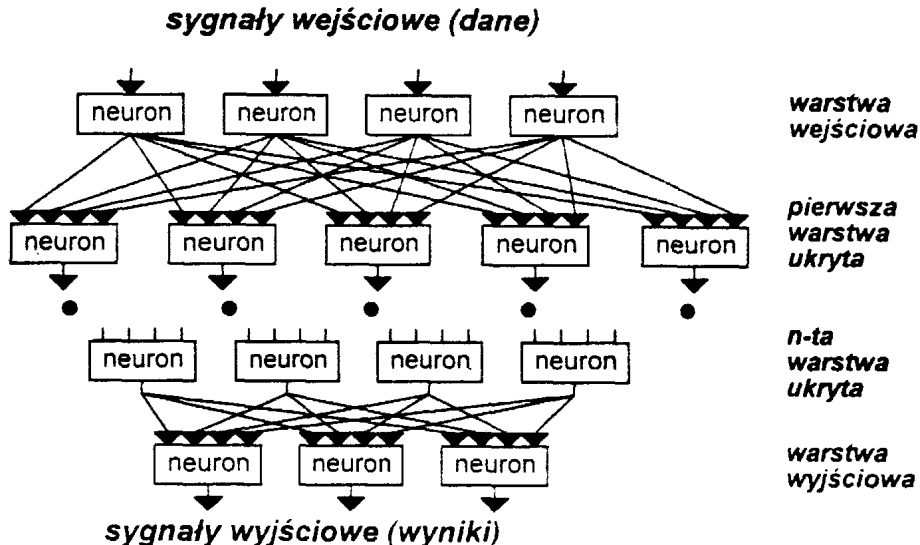
Możemy zauważyć, że pierwsze trzy funkcje przedstawione w tabeli są nieróżniczkowalne. Cecha ta może utrudniać proces uczenia się sieci neuronowej (omówione dokładniej w podrozdziale 1.4.2)), dlatego często stosuje się ich przybliżenia (funkcje sigmoidalne oraz *softmax*). W celu osiągnięcia założen niniejszej pracy jako funkcję aktywacji neuronu przyjęto funkcję *ReLU*.

Po zdefiniowaniu funkcji wejściowej oraz aktywacji ostateczną odpowiedź neuronu nieliniowego możemy zapisać jako

$$y = \varphi(f_{\vec{w}}(\vec{x})).$$

### 1.3 Schemat prostej sieci neuronowej

Klasyczną strukturę sieci neuronowej możemy podzielić na warstwy. Jej schemat przedstawiony jest na poniższym rysunku<sup>3</sup>.



Rysunek 1.3: Schemat klasycznej sieci neuronowej.

Pierwszą warstwę, do której docierają impulsy ze środowiska, nazywamy warstwą wejściową, natomiast ostatnia nazywana jest warstwą wyjściową. Między nimi mogą występować tzw. warstwy ukryte. Odpowiedzi neuronów warstwy wyjściowej są ostateczną odpowiedzią sieci na otrzymane dane wejściowe. Może być ona interpretowana jako np. przypisanie badanego obiektu do danej klasy lub, jak w przypadku rozpatrywanego przez nas problemu, instrukcje wysyłane do sterowania modelem lub robotem.

<sup>3</sup>Źródło: R. Tadeusiewicz, T. Gąciarz, B. Borowik, B. Leper - "Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#"

Zazwyczaj na początkowym etapie tworzenia sieci każdy neuron danej warstwy jest połączony z każdym neuronem następnej warstwy. Dopiero w trakcie procesu uczenia sieć „sama” dezaktywuje niektóre neurony (poprzez minimalizację przypisanych im wag).

Sieci neuronowe, posiadające neurony nieliniowe, mają ogromne możliwości w odwzorowywaniu funkcji. Sieć z tylko jedną warstwą ukrytą może posłużyć do zaimplementowania dowolnej funkcji boolowskiej, którą możemy zapisać jako:

$$f : \{0, 1\}^n \rightarrow \{0, 1\},$$

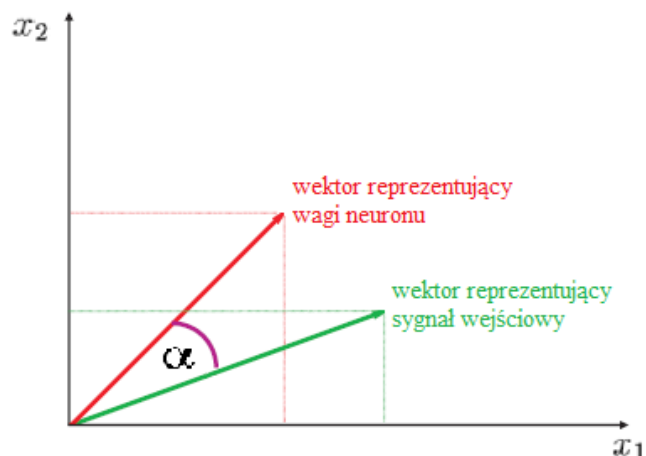
dla dowolnie dużego  $n$ . Ponadto, jeśli stosujemy funkcję aktywacji, która jest monotoniczna oraz ciągła, możemy skonstruować sieć neuronową będącą w stanie przybliżyć wartość dowolnej funkcji rzeczywistej z dowolną dokładnością.

W niniejszej pracy będziemy badać zachowanie sieci posiadających jedną lub dwie warstwy ukryte.

## 1.4 Proces uczenia sieci

### 1.4.1 Podstawy uczenia sieci neuronowej

Rozpatrzmy przypadek, gdy każdy neuron w sieci ma dwa wejścia  $x_1$  i  $x_2$  oraz odpowiadające im wagi  $w_1$  i  $w_2$ . Wtedy wektor sygnału wejściowego oraz wektor wag neuronu możemy przedstawić na płaszczyźnie tak, jak zostało to pokazane na poniższej ilustracji <sup>4</sup>.



Rysunek 1.4: Wzajemne położenie wektora sygnałów wejściowych oraz wektora wag neuronu.

Przez  $\alpha$  oznaczmy kąt między wektorem sygnału wejściowego oraz wektorem wag neuronu. Wtedy funkcję wejściową dla rozpatrywanego przypadku możemy zapisać następująco:

$$f_{\vec{w}}(\vec{x}) = x_1 w_1 + x_2 w_2,$$

co jest równoważne

$$f_{\vec{w}}(\vec{x}) = |\vec{x}| \cdot |\vec{w}| \cdot \cos \alpha.$$

Możemy zauważyć, że dla znormalizowanych wektorów sygnału wejściowego oraz wag neuronu, funkcja wejściowa neuronu da tym większy wynik, im mniejszy był kąt  $\alpha$ .

W dalszej części pracy będziemy posługiwać się pojęciem siły, z jaką neuron zareagował na dany sygnał wejściowy. Oznacza to, jak duży wynik zwraca funkcja wejściowa neuronu. Im wagi neuronu są bardziej zbliżone do odpowiadających im składowych wektora sygnału wejściowego, tym neuron silniej reaguje na taki

<sup>4</sup>Źródło: R. Tadeusiewicz, T. Gąciarz, B. Borowik, B. Leper - "Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#"

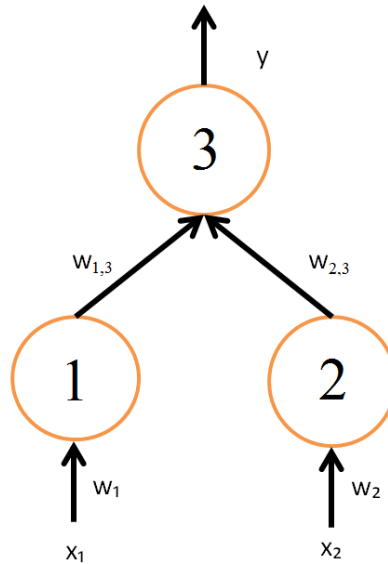


sygnał.

Proces uczenia się sieci samoorganizujących się (umówiony dalej w podrozdziale 1.4.5) możemy przedstawić jako ciąg akcji mających na celu zmniejszenie różnicy między wektorem wag neuronu i wektorem sygnału wejściowego.

### 1.4.2 Propagacja wsteczna

Rozpatrzmy prostą sieć neuronową przedstawioną na poniższej ilustracji<sup>5</sup>:



Rysunek 1.5: Przykład prostej sieci neuronowej.

Początkowe wagi neuronów są generowane w sposób losowy. Dla danego sygnału wejściowego  $\vec{x} = (x_1, x_2)$  sieć generuje odpowiedź  $y$ . Załóżmy, że oczekiwaliśmy od sieci, że wygeneruje odpowiedź  $y'$ . Przez  $\Delta$  oznaczmy tzw. *funkcję straty*, która będzie mierzyła, jak bardzo oczekiwana przez nas odpowiedź  $y'$  różni się od tej aktualnie zwróconej przez sieć ( $y$ ). Często przyjmuje ona postać

$$\Delta = (y' - y)^2.$$

Dla powyższej sytuacji proces uczenia sieci możemy przedstawić jako próbę takiego dobrania wag, aby zminimalizować wartość  $\Delta$ .

Dla neuronu oznaczonego numerem 3 mamy wszystkie potrzebne dane, aby dokonać zmian jego wag. Oznaczmy różnicę między odpowiedzią tego neuronu i oczekiwanym rezultatem jako  $\Delta_3 = (y' - y)^2$ . Możemy zatem odpowiednio zmodyfikować wartości wag  $w_{1,3}$  oraz  $w_{2,3}$  aby zminimalizować wartość  $\Delta_3$ . Jednak nie znamy wartości  $\Delta$  dla neuronów 1 oraz 2. Wartości te otrzymamy w wyniku zastosowania tzw. zasady propagacji wstecznej. Możemy zauważyć, że zarówno neuron 1 jak i 2 wpłynęły na ostateczną wartość  $\Delta_3$  wprost proporcjonalnie do pierwotnych wartości wag  $w_{1,3}$  oraz  $w_{2,3}$ . Dlatego ich wartości  $\Delta$  będą również zależały od tych wag. Możemy zatem przyjąć wartości  $\Delta_1 = \Delta_3 * w_{1,3}$  oraz  $\Delta_2 = \Delta_3 * w_{2,3}$  odpowiednio dla neuronu 1 i 2. Ostatecznie, na podstawie  $\Delta_1$  i  $\Delta_2$ , modyfikujemy wagi  $w_1$  i  $w_2$ .

Uogólniając, możemy powiedzieć, że wartości  $\Delta$  dla neuronów z  $k$ -tej warstwy wyliczane są według zasady propagacji wstecznej na podstawie wartości  $\Delta$  z warstwy  $k + 1$ .

Przedstawiony proces propagacji wstecznej odbywa się w pętli, aż do momentu osiągnięcia przyjętego warunku zbieżności (np. gdy wartość *funkcji straty* dla kolejnych  $y$  i  $y'$  będzie mniejsza od ustalonego  $\epsilon$ ).

<sup>5</sup><https://en.wikipedia.org/wiki/Backpropagation>

### 1.4.3 Uczenie z nauczycielem

Do wytrenowania sieci niezbędny jest dostatecznie duży zbiór danych wejściowych. W przypadku uczenia sieci z nauczycielem (ang. supervised learning) odpowiada im również zbiór oczekiwanych wyników. Każdy etap procesu uczenia możemy podzielić na następujące kroki:

- do sieci trafia sygnał wejściowy,
- każdy neuron na podstawie przypisanych mu wag i przyjętej funkcji aktywacji (jeśli jest to neuron nieliniowy) oblicza własny sygnał wyjściowy,
- ostateczna odpowiedź sieci jest porównywana z oczekiwanym wynikiem,
- stosowany jest algorytm propagacji wstecznej (omówiony w podrozdziale 1.4.2) w celu zmodyfikowania wartości wag neuronów.

### 1.4.4 Samouczenie sieci neuronowej

W przypadku samouczenia się sieci neuronowej (ang. unsupervised learning) zbiór danych uczących składa się jedynie z danych wejściowych. Mimo braku zbioru oczekiwanych wyników, przypisanych do konkretnych sygnałów wejściowych, przy zastosowaniu odpowiednich metod uczenia się neurony są w stanie nauczyć się klasyfikować otrzymywane dane na pewne klasy i odkrywać zachodzące między nimi zależności. Przykładem takiej metody uczenia się jest metoda Hebb'a. Możemy ją przedstawić jako proces, którego każdy krok wygląda następująco:

- do sieci trafia sygnał wejściowy,
- każdy neuron oblicza swoją odpowiedź na otrzymane dane,
- wszystkie wagi zmieniają się według reguły  $w'_i = w_i + \Delta w_i$ .

W powyższym wzorze  $w'_i$  oznacza nową wartość  $i$ -tej wagi neuronu, natomiast  $\Delta w_i$  jest nazywana *przyrostem wagi*. Powstało wiele reguł mówiących, jak należy dobierać  $\Delta w_i$ . Jedną z nich jest *prosta reguła Hebb'a*:

$$\Delta w_i = \eta \cdot x_i \cdot y,$$

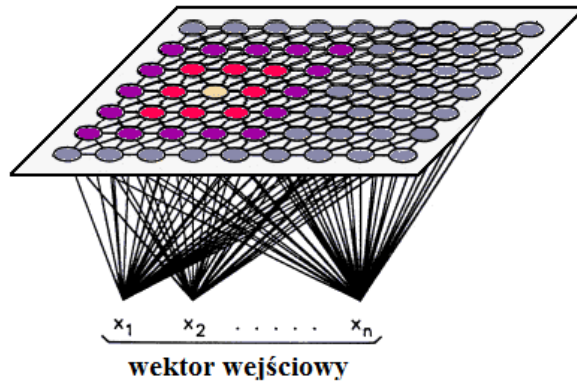
gdzie  $\eta$  jest nazywana *współczynnikiem proporcjonalności*, który wpływa na wielkość wprowadzanych zmian.

### 1.4.5 Sieci samoorganizujące się

Struktura sieci samoorganizującej się jest nieco inna od typowej struktury sieci neuronowej (przedstawionej w podrozdziale 1.3). Możemy ją przedstawić jako graf, w którym węzły symbolizują neurony, natomiast krawędzie oznaczają sąsiedztwo między nimi. Przykładem takiej sieci z neuronami znajdującymi się w dwóch wymiarach jest *sieć Kohonena*. Została ona przedstawiona na poniższej ilustracji <sup>6</sup>.

---

<sup>6</sup>Źródło: R. Tadeusiewicz, T. Gąciarz, B. Borowik, B. Leper - "Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#"



Rysunek 1.6: Schemat sieci Kohonena.

Proces uczenia się takiej sieci również różni się od uprzednio przedstawionych. Dla każdego sygnału wejściowego wyłaniany jest tzw. zwycięzca (ang. competitive learning). Jest to neuron, który najsilniej zareagował na podane dane wejściowe. Następnie wagi wszystkich neuronów są modyfikowane zgodnie z przyjętą metodą samouczenia. Jedną z takich metod może być omówiona w podrozdziale 1.4.4 *metoda Hebb*. Zgodnie z jej założeniami *przyrost wagi* jest zależny od *współczynnika proporcjonalności*  $\eta$ . Współczynnik ten będzie największy dla zwycięskiego neuronu i będzie malał wraz ze wzrostem odległości od niego. Zatem w każdym kroku procesu uczenia się najsilniej zostaną zmodyfikowane wagi neuronu zwycięskiego i jego najbliższych sąsiadów. Po zakończeniu procesu uczenia w sieci powstaną grupy neuronów (tzw. klastry) identyfikujące podobne wektory wejściowe.

### 1.4.6 Uczenie się ze wzmocnieniem

Czasami wygodnie jest operować modelem, w którym mamy zdefiniowany zbiór stanów środowiska  $S$  oraz zbiór możliwych do podjęcia akcji  $A$ . Wykonanie każdej akcji powoduje zmianę stanu środowiska. Proces uczenia ze wzmocnieniem (ang. reinforcement learning) możemy przedstawić jako cykl, w trakcie którego algorytm nabywa zdolność wybrania dla danego stanu najlepszej, pozwalającej na osiągnięcie ogólnie przyjętego celu, akcji. Każdy krok  $t$  takiego procesu wygląda następująco<sup>7</sup>:

- obserwuj aktualny stan  $s_t$ ,
- wybierz akcję  $a_t$ , do wykonania w stanie  $s_t$ ,
- wykonaj akcję  $a_t$ ,
- obserwuj wzmocnienie  $r_t$  i następny stan  $s_{t+1}$ ,
- ucz się na podstawie doświadczenia  $\langle s_t, a_t, r_t, s_{t+1} \rangle$ .

Przykładem algorytmu implementującego uczenie się ze wzmocnieniem jest *Q-learning*. Algorytm ten generuje funkcję  $Q$ , która dla danego stanu  $S$  i akcji  $A$  zwraca liczbę rzeczywistą (nagrodę):

$$Q : S \times A \rightarrow \mathbb{R}.$$

Aktualizację wartości funkcji  $Q$  przedstawia poniższa reguła:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \left( r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right),$$

<sup>7</sup><http://osilek.mimuw.edu.pl/index.php?title=Sztuczna%20inteligencja/SI%20Modu%C5%82%2013%20-%20Uczenie%20si%C4%99%20ze%20wzmocnieniem>

gdzie:

- $Q(s_t, a_t)$  - stara wartość,
- $\alpha$  - szybkość uczenia się,
- $r_{t+1}$  - nagroda,
- $\gamma$  - współczynnik dyskontowy,
- $\max_a Q(s_{t+1}, a)$  - szacowana przyszła wartość.

W procesie generowania funkcji  $Q$  można przyjąć różne strategie wyboru akcji. Jedną z nich jest strategia  $\epsilon$ -zachłanna. Według niej, dla uprzednio dobranej wartości  $\epsilon$ , w każdym kroku procesu generowania funkcji z prawdopodobieństwem równym  $\epsilon$  wybieramy akcję losową, natomiast z prawdopodobieństwem  $1 - \epsilon$  wybieramy akcję wskazaną przez funkcję  $Q$  (tę, dla której  $Q$  zwróciła największą wartość).

Sieci neuronowe doskonale nadają się do zaimplementowania funkcji  $Q$ . Sieć taka posiada liczbę neuronów w warstwie wyjściowej odpowiadającą liczbie możliwych do podjęcia akcji. Dla danego stanu  $s_t$  generowane są wartości dla każdej akcji, a ta z największą wartością powinna zostać wykonana w kroku  $t$ . Aktualizacja wartości funkcji  $Q$  polega na odpowiednim dopasowaniu wartości wag neuronów za pomocą propagacji wstecznej.

W niniejszej pracy w procesie uczenia się sieci neuronowej zastosujemy właśnie algorytm  $Q$ -learning. Ponadto przyjętą strategią wyboru akcji będzie strategia  $\epsilon$ -zachłanna.



# Przedstawienie narzędzi wybranych do realizacji projektu

Do realizacji projektu zastosowano następujące narzędzia:

- Unity – jest to platforma pozwalająca na tworzenie modeli 3D oraz animacji. Zaimplementowano w niej środowisko symulujące rzeczywiste zjawiska fizyczne, co pozwoliło na przeprowadzenie procesu uczenia sieci neuronowej w warunkach zbliżonych do realnych. Platforma ta została wybrana ze względu na swoją ogromną popularność oraz łatwość, z jaką można konstruować i modyfikować w niej projekty,
- .NET Framework – środowisko uruchomieniowe dla skryptów napisanych w języku C# (wszystkie programy obsługiwane przez Unity muszą być napisane w tym języku),
- biblioteki Theano oraz numpy języka python - pozwalają na przeprowadzanie operacji matematycznych na wielowymiarowych tablicach,
- biblioteka Keras języka python (bazująca na bibliotece Theano) - implementuje sieci neuronowe. Pozwala ona na bardzo szybką konstrukcję i modyfikację sieci neuronowej z jednoczesnym bardzo wydajnym przeprowadzaniem obliczeń. Umożliwiło to szybkie generowanie sygnałów sterujących modelem oraz zbadanie wpływu struktury sieci neuronowej na wyniki końcowe programu,
- biblioteka pickle języka python - umożliwia serializację obiektów i ich zapis do pliku. Dzięki niej możliwe było zapisanie w pliku utworzonej sieci neuronowej wraz z wartościami wag poszczególnych neuronów,
- biblioteka watchdog języka python - pozwala na monitorowanie plików (np. sprawdzanie, czy na danym pliku nie są w danym momencie wykonywane żadne operacje).

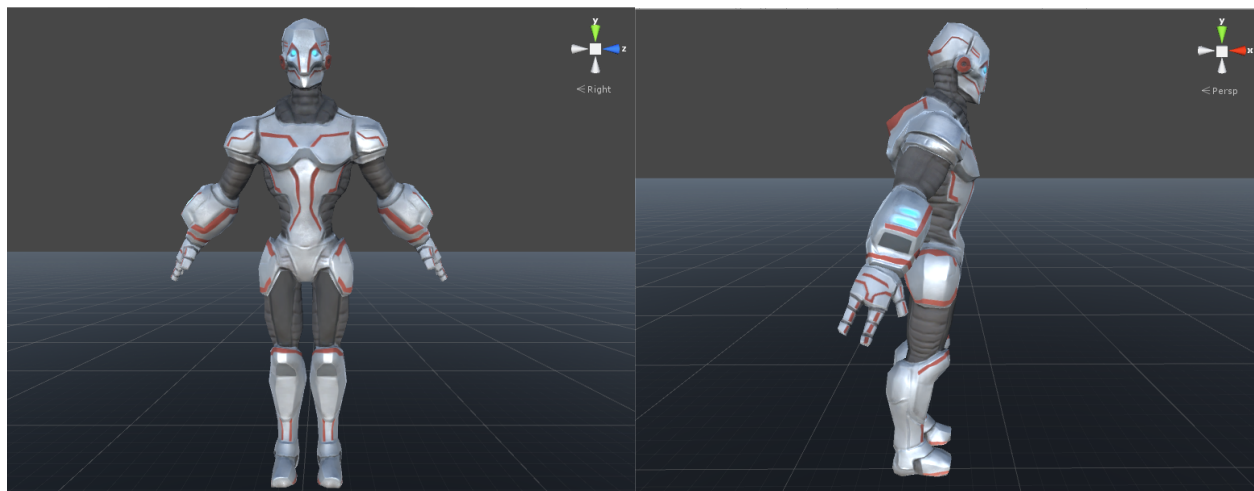




# Opis implementacji modelu komputerowego oraz sieci neuronowej

## 3.1 Schemat modelu komputerowego

Stworzony na potrzeby tej pracy model symuluje zachowanie robota humanoidalnego. Został on przedstawiony na poniższych ilustracjach.



Rysunek 3.1: Model komputerowy wykorzystywany w pracy.

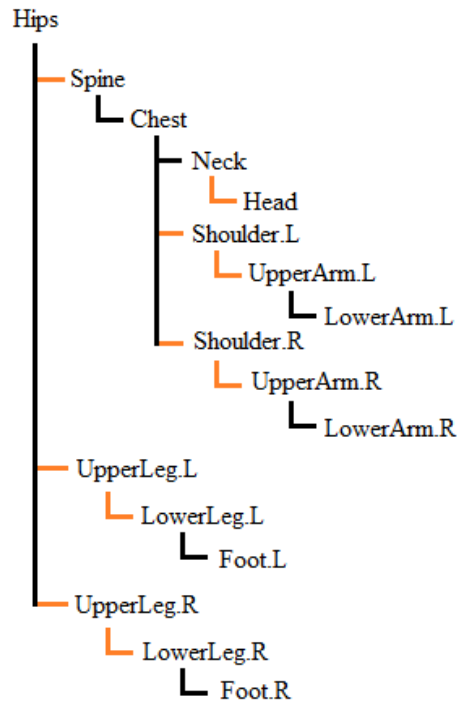
Występujące w nim części oraz zależności między nimi zostały przedstawione na schemacie na następnej stronie.

Do każdej z wymienionych części przypisany jest tzw. *zderzak* (ang. *collider*) oraz skrypt *OnCollision.cs*, który zostanie omówiony w podrozdziale 3.3. *Zderzaki* pozwalają na interakcję modelu z otoczeniem (zapobiegają jego przenikaniu przez podłoże i informują o zderzeniach różnych obiektów).

Na schemacie zaznaczono również kolorem pomarańczowym miejsca występowania stawów (nazywane w Unity z ang. *CharacterJoints*). Mają one nałożone ograniczenia ruchu mające na celu jak najwierniejsze oddanie ruchu prawdziwego człowieka.

Mimo występowania w modelu tak wielu części w procesie uczenia sieci neuronowej będziemy poruszać tylko pięcioma z nich. Są to: *Spine*, *UpperLeg.L*, *LowerLeg.L*, *UpperLeg.R* oraz *LowerLeg.R*. Do każdej z nich przypisany jest skrypt *MoveJoint.cs*, który tak jak *OnCollision.cs* zostanie dokładnie opisany w podrozdziale 3.3.

Unity do wykonywania symulacji używa tzw. scen, które z kolei dzielą się na występujące po sobie kratki. Przed uruchomieniem danej sceny możemy dowolnie modyfikować położenie umieszczonych w niej obiektów. Uruchomienie sceny powoduje rozpoczęcie symulowania zjawisk fizycznych (za pomocą tzw. *UnityEngine*) oraz zainicjowanie wszystkich powiązanych z nią skryptów. Scena używana w naszym projekcie składa się jedynie z omówionego modelu oraz płaskiego podłoża.



Rysunek 3.2: Schemat części modelu i zależności między nimi.

## 3.2 Zarys działania programu

Zaimplementowana w niniejszej pracy sieć neuronowa może działać w dwóch trybach - w trybie uczenia się oraz generowania sygnałów sterujących modelem. Zostały one omówione w następnych podrozdziałach.

### 3.2.1 Sterowanie modelem

W każdym kroku działania programu będziemy odczytywać następujące dane o modelu:

- wysokość *Spine* nad podłożem,
- odchylenie *Spine* od osi pionowej,
- położenie *UpperLeg.L*, *LowerLeg.L*, *UpperLeg.R* oraz *LowerLeg.R* względem *Spine*,
- dystans przebyty od miejsca początkowego.

Mimo tego, że model został stworzony w trzech wymiarach, odczytywane dane o położeniu modelu będą pochodziły tylko z dwóch wymiarów  $X$  i  $Y$ .

Powyższe dane możemy rozumieć jako *stan*, w którym aktualnie znajduje się model. Zbiór wszystkich takich *stanów* oznaczmy przez  $S$ .

Ponadto, dla każdej części, do której został przypisany skrypt *MoveJoint* zostaną stworzone dwie metody (*akcje*) sterujące (odpowiadające ruchom do przodu i do tyłu). Ponieważ mamy pięć takich części, otrzymamy dziesięć metod sterujących. Zbiór *akcji*  $A$  możemy zapisać w postaci

$$A = (a_1, a_2, a_3, \dots, a_8, a_9, a_{10}).$$

Stworzona sieć neuronowa w swojej warstwie wyjściowej będzie posiadała dziesięć neuronów (będzie zwracała dziesięć różnych liczb rzeczywistych). Możemy zatem powiedzieć, że sieć implementuje następującą funkcję:

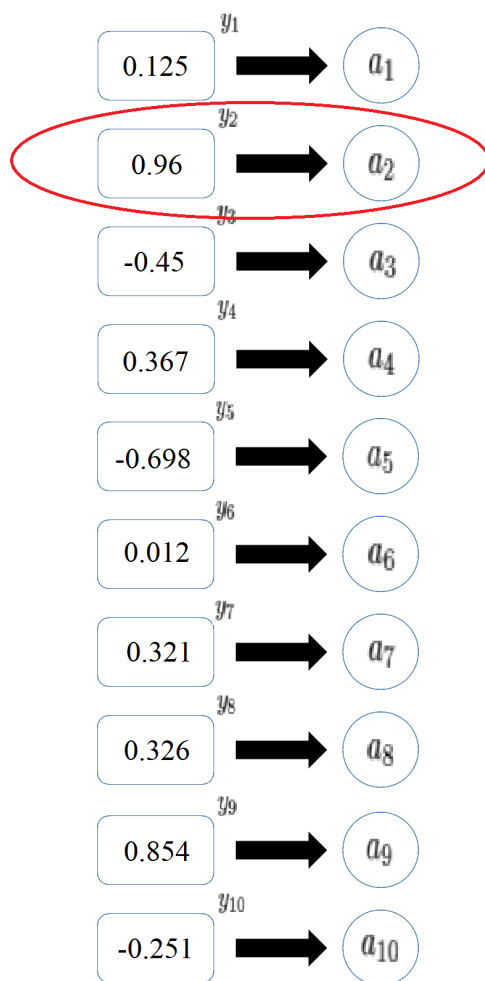
$$f : S \rightarrow \mathbb{R}^{10}.$$

Odpowiedź sieci oznaczmy jako  $\vec{y}$ . Możemy ją zapisać jako

$$\vec{y} = (y_1, y_2, y_3, \dots, y_8, y_9, y_{10}).$$

Ponadto, każdy neuron warstwy wyjściowej będzie miał przypisaną do siebie metodę (*akcję*). Zatem do każdej składowej wektora  $\vec{y}$  będzie przypisana *akcja* ze zbioru  $A$  ( $a_1$  odpowiada  $y_1$ ,  $a_2$  odpowiada  $y_2$  itd.).

Jednak dla każdego kroku procesu działania programu będziemy mogli wybrać tylko jedną *akcję*. Spośród wszystkich składowych wektora  $\vec{y}$  wybierana będzie ta, która ma największą wartość i wykonywana będzie odpowiadająca jej *akcja*. Poniższa ilustracja przedstawia przykład sygnału wyjściowego sieci oraz wybraną dla niego *akcję*.



Rysunek 3.3: Przykład sygnału wyjściowego wraz z odpowiadającymi mu akcjami.

Możemy zatem powiedzieć, że sieć neuronowa odzwierciedla funkcję przypisującą każdemu *stanowi*, w którym może się znaleźć model, daną *akcję*:

$$f : S \rightarrow A.$$



### 3.2.2 Proces uczenia się sieci neuronowej

Za metodę odpowiednią do realizacji zadań niniejszej pracy uzanano technikę *Q-learning* uczenia ze wzmocnieniem (omówiona w podrozdziale 1.4.6). Ponadto przyjęta została w nim strategia  $\epsilon$ -zachłanna. Przed dokładnym omówieniem tego procesu musimy jeszcze zdefiniować tzw. *funkcję nagrody*. Jest to funkcja, która dla danego stanu modelu zwraca liczbę, będącą nagrodą w danym kroku. *Funkcja nagrody* w naszym programie bierze pod uwagę następujące warunki:

- wychylenie *Spine* od pionu jest w zakresie od  $2^\circ$  do  $30^\circ$ ,
- aktualna przebyta od miejsca początkowego odległość jest większa od odległości z poprzedniego kroku procesu uczenia,
- wysokość *Spine* nad podłożem wynosi przynajmniej 80% wysokości modelu.

Jeśli tylko dwa pierwsze warunki zostały spełnione, funkcja zwraca wartość 5. Jeśli został spełniony również trzeci warunek, funkcja zwraca wartość 10. Ponadto, jeśli model przesuwa się do tyłu, funkcja nagrody zwraca liczbę -5. W pozostałych przypadkach zwracana jest wartość -1.

Celem, który ma osiągnąć sieć neuronowa, jest maksymalizowanie wartości otrzymywanych nagród.

Dla każdego otrzymanego sygnału wejściowego sieć oblicza swój sygnał wyjściowy, na podstawie którego wyłaniania jest odpowiednia *akcja*. Po wykonaniu danej *akcji* przez model, funkcja nagrody zwraca liczbę będącą nagrodą (na podstawie położenia, w jakim znajduje się model). Wartość ta jest następnie podstawiana pod odpowiedź neuronu, z którym skojarzona jest wykonana *akcja*. Ostatecznie, wartości wag sieci neuronowej są zmieniane zgodnie z zasadą propagacji wstecznej, uwzględniając zmodyfikowaną odpowiedź sieci.

Aby zobrazować powyższy proces, prześledźmy następujący przykład - do sieci trafił sygnał wejściowy, na podstawie którego sieć wygenerowała odpowiedź

$$\vec{y} = (0.12, 0.45, -0.12, 0.432, 2.45, 5.67, -3.21, 0.512, 0.01, 0.421).$$

Maksymalna wartość w powyższym wektorze wynosi 5.67, dlatego model wykona *akcję*  $a_6$ . Po jej zakończeniu *funkcja nagrody* zwróciła wartość 10. Wartość tę podstawiamy w miejsce poprzedniej 5.67, zatem nasz sygnał wyjściowy wygląda teraz następująco:

$$\vec{y}' = (0.12, 0.45, -0.12, 0.432, 2.45, \mathbf{10}, -3.21, 0.512, 0.01, 0.421).$$

Ostatecznie, na podstawie sygnału  $\vec{y}'$ , wagi neuronów sieci są modyfikowane zgodnie z zasadą propagacji wstecznej.

Powiedzieliśmy również, że przedstawiony algorytm korzysta ze strategii  $\epsilon$ -zachłannej. Oznacza to, że w każdym kroku procesu uczenia wybierana *akcja* będzie z prawdopodobieństwem równym przyjętemu  $\epsilon$  *akcją* losową, natomiast z prawdopodobieństwem  $1 - \epsilon$  tą, którą wskazała sieć. Prześledźmy ten proces na powyższym przykładzie. Sieć zwróciła taką samą odpowiedź  $\vec{y}$ . Przyjmijmy, że  $\epsilon = 0.98$ . Następnie losujemy liczbę  $p$  z zakresu od 0.0 do 1.0. Jeśli  $p$  będzie mniejsza od przyjętego  $\epsilon$ , wybierzemy *akcję* losową. W przeciwnym wypadku wybierzemy *akcję* wskazaną przez sieć.

Załóżmy, że wylosowana  $p$  wynosi 0.5. Zatem wybieramy *akcję* zupełnie losową. Przyjmijmy, że jest to *akcja*  $a_2$ . Po jej wykonaniu funkcja nagrody ponownie zwróciła wartość 10. Zatem zmodyfikowana odpowiedź sieci wygląda teraz następująco:

$$\vec{y}' = (0.12, \mathbf{10}, -0.12, 0.432, 2.45, 5.67, -3.21, 0.512, 0.01, 0.421).$$

Po otrzymaniu powyższego wektora ponownie modyfikujemy wagi zgodnie z zasadą propagacji wstecznej.

W naszym programie początkowa wartość  $\epsilon$  będzie wynosiła 1.0 i w trakcie procesu uczenia się sieci będzie stopniowo malała. Gdy osiągnie ona wartość 0.1, proces uczenia zostanie zakończony, a sieć zostanie uruchomiona w trybie generowania sygnałów sterujących. Możemy zatem zauważyć, że początkowo *akcje* będą wybierane w sposób losowy, natomiast wraz z biegiem czasu coraz więcej z nich będzie wybieranych przez sieć aż do momentu, gdy cały proces sterowania będzie polegał na wskazaniach sieci.

### 3.3 Kod źródłowy najważniejszych części projektu

Najważniejsze skrypty, z których składa się projekt, to:

- OnCollision.cs,
- MoveJoint.cs,
- ModelController.cs,
- NeuralNetwork.py.

Wszystkie zostaną dokładnie omówione w następnych podrozdziałach.

#### 3.3.1 OnCollision

Kod źródłowy tego skryptu wygląda następująco:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class OnCollision : MonoBehaviour {

    void OnCollisionEnter(Collision collision) {
        if (collision.collider.gameObject.name == "Ground") {
            touchingFloor = true;
            if (this.name.Equals("Spine") || this.name.Equals("Chest") ||
                this.name.Equals("Neck") || this.name.Equals("Head") ||
                this.name.Equals("Shoulder.R") ||
                this.name.Equals("Shoulder.L") ||
                this.name.Equals("UpperArm.R") ||
                this.name.Equals("UpperArm.L")) {

                Scene scene = SceneManager.GetActiveScene();
                SceneManager.LoadScene(scene.name);
            }
        }
    }
}
```

Biblioteki *UnityEngine* oraz *UnityEngine.SceneManagement* pozwalają na używanie silnika Unity oraz scen, które zostały omówione w sekcji 3.1. Możemy zauważyć, że klasa *OnCollision* dziedziczy po klasie *MonoBehaviour*. Klasa *MonoBehaviour* jest bazową klasą w Unity, po której dziedziczą wszystkie skrypty. Pozwala ona na korzystanie z takich metod jak *Start()* i *Update()*, które zostaną dokładniej omówione podczas opisywania poszczególnych skryptów.

Metoda *OnCollisionEnter* jest wywoływana za każdym razem gdy dany *zderzak* dotknie innego *zderzaka*. Jeśli częścią modelu, która dotknęła podłoża, jest *Spine*, *Chest*, *Neck*, *Head*, *Shoulder.R*, *Shoulder.L*, *UpperArm.R* lub *UpperArm.L* (stan, w którym model upadł) scena zostaje zresetowana. Kontynuowanie procesu uczenia sieci w takiej sytuacji mogłoby negatywnie wpłynąć na końcowe działanie programu.



### 3.3.2 MoveJoint

```
using UnityEngine;

public class MoveJoint : MonoBehaviour {
    public float minXTiltAngle = 10.0F;
    public float maxXTiltAngle = 10.0F;
    public float minYTiltAngle = 10.0F;
    public float maxYTiltAngle = 10.0F;
    public float minZTiltAngle = 10.0F;
    public float maxZTiltAngle = 10.0F;
    public float smooth = 10.0F;

    Transform jointTransform = null;
    GameObject parent = null;

    private Quaternion previousPosition;

    private float initialLocalRotationX;
    private float initialLocalRotationY;
    private float initialLocalRotationZ;
    ...
}
```

Pola *minXTiltAngle*, *maxXTiltAngle* itd. odpowiadają minimalnemu oraz maksymalnemu kątowi, o który rozpatrywana część modelu może się wychylić w danym wymiarze. Początkowo pola te przyjmują wartość 10.0, jednak Unity pozwala na ich ręczne modyfikowanie już w samej platformie, co nie wpływa na kod źródłowy. Zatem różne części modelu, w zależności od pełnionych funkcji, będą przyjmowały różne wartości tych pól.

Pole *smooth* wpływa na szybkość, z jaką będzie poruszać się dana kończyna. Dla całego modelu przyjmuje ona wartość 10.0.

Pole *jointTransform* jest obiektem klasy *Transform*, która przechowuje informacje o położeniu i rotacji danego obiektu oraz pozwala na ich manipulację. *jointTransform* zostanie dalej przypisany do stawu wchodzącego w skład danej części modelu.

*GameObject* jest klasą bazową, po której dziedziczą wszystkie jednostki w Unity. Do pola *parent*, będącego obiektem tej klasy, w dalszej części programu zostanie przypisany obiekt nadrzędny (rodzic) danego obiektu (np. dla *LowerLeg.R* będzie to *UpperLeg.R*).

Do reprezentowania rotacji Unity wykorzystuje obiekty klasy *Quaternion*. Pole *previousPosition* tej klasy będzie przechowywało ostatnią rotację danej części modelu.

Pola *initialLocalRotationX* itd. będą przechowywały rotację danej części modelu względem rodzica z momentu uruchamiania sceny.

```
public void Start () {
    CharacterJoint characterJoint =
        gameObject.GetComponent(typeof(CharacterJoint)) as CharacterJoint;
    if (characterJoint != null) {
        jointTransform = characterJoint.transform;
    }
    initialLocalRotationX = gameObject.transform.localEulerAngles.x;
    initialLocalRotationY = gameObject.transform.localEulerAngles.y;
    initialLocalRotationZ = gameObject.transform.localEulerAngles.z;

    parent = jointTransform.parent.gameObject;
}
```

Metoda *Start()* jest uruchamiana tylko jeden raz podczas inicjalizowania całej sceny. Dla danej części modelu metoda ta wykonuje następujące czynności:

- do pola *characterJoint* zostaje przypisany jej staw (jeśli taki staw w niej występuje),
- zapamiętane są jej początkowe rotacje,
- do pola *parent* przypisany zostaje jej rodzic.

```
public void Move(float tiltAroundX, float tiltAroundY, float tiltAroundZ){
    bool moveX = true;
    bool moveY = true;
    bool moveZ = true;
    float currentRotationX =
        Mathf.Abs(gameObject.transform.localEulerAngles.x -
            initialLocalRotationX);
    float currentRotationY =
        Mathf.Abs(gameObject.transform.localEulerAngles.y -
            initialLocalRotationY);
    float currentRotationZ =
        Mathf.Abs(gameObject.transform.localEulerAngles.z -
            initialLocalRotationZ);

    if (*) {
        moveX = false;
    }
    if (*) {
        moveY = false;
    }
    if (*) {
        moveZ = false;
    }

    if (tiltAroundX <= 0) tiltAroundX = tiltAroundX * minXTiltAngle;
    else tiltAroundX = tiltAroundX * maxXTiltAngle;

    if (tiltAroundY <= 0) tiltAroundY = tiltAroundY * minYTiltAngle;
    else tiltAroundY = tiltAroundY * maxYTiltAngle;

    if (tiltAroundZ <= 0) tiltAroundZ = tiltAroundZ * minZTiltAngle;
    else tiltAroundZ = tiltAroundZ * maxZTiltAngle;

    float rotationX = gameObject.transform.localEulerAngles.x;
    float rotationY = gameObject.transform.localEulerAngles.y;
    float rotationZ = gameObject.transform.localEulerAngles.z;

    Quaternion target = new Quaternion();
    if (moveX) rotationX = initialLocalRotationX - tiltAroundX;
    if (moveY) rotationY = initialLocalRotationY - tiltAroundY;
    if (moveZ) rotationZ = initialLocalRotationZ - tiltAroundZ;

    target = parent.transform.rotation *
        Quaternion.Euler(rotationX, rotationY, rotationZ);
}
```





```
jointTransform.rotation =
    Quaternion.Slerp(transform.rotation, target, Time.deltaTime*smooth);
}
```

Metoda *Move()* przyjmuje trzy argumenty: *tiltAroundX*, *tiltAroundY* oraz *tiltAroundZ*. Każdy z nich jest typu float i przyjmuje wartości z zakresu od -1.0 do 1.0.

W metodzie *Move()* występują trzy zmienne boolowskie *moveX*, *moveY* i *moveZ*. Każda z nich przyjmuje wartość *true*, jeśli w danym wymiarze ma nastąpić ruch części modelu. Każda z nich jest inicjowana wartością *true*.

Zmienne *currentRotationX*, *currentRotationY* oraz *currentRotationZ* przechowują wartości o aktualnej rotacji obiektu (jest to wartość bezwzględna różnicy aktualnej rotacji obiektu względem rodzica i początkowej rotacji względem rodzica).

Wnętrza warunków *if (\*)* ... zostały pominięte, aby zwiększyć czytelność kodu źródłowego. Jest w nich porównywana aktualna rotacja obiektu w danym wymiarze z maksymalną możliwą do osiągnięcia. Jeśli rotacja ta osiągnęła maksimum, zmienna *move*, odpowiadająca danemu wymiarowi, przyjmuje wartość *false*. Dzięki temu zabiegowi obiekt nie przekroczy nałożonych ograniczeń ruchu w danej osi.

Następnie sprawdzane są wartości argumentów przyjmowanych przez metodę *Move()*. Jeśli wartości te są mniejsze od 0.0, są one przemnażane przez odpowiednie *minTiltAngle*, a jeśli większe, to przez *maxTiltAngle*. Zatem, jeśli argument jest mniejszy od 0.0, ruch wykonywany w stronę ograniczoną przez kąt *minTiltAngle*, a jeśli większy od 0.0, to w stronę ograniczoną przez *maxTiltAngle*.

Pod zmiennymi *rotationX*, *rotationY* oraz *rotationZ* zapamiętywana jest aktualna rotacja obiektu. Jest to konieczne w przypadku, gdy został osiągnięty maksymalny zakres ruchu w danej osi. Będą one potrzebne w momencie inicjalizowania ruchu.

Dalej tworzony jest nowy obiekt *target* klasy *Quaternion*, który będzie docelową rotacją danej części modelu.

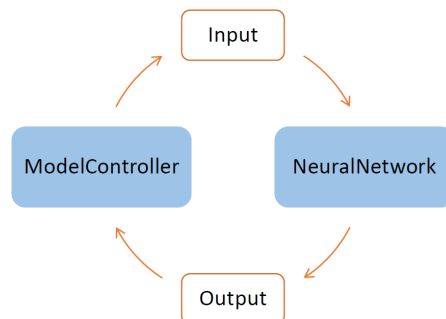
Jeśli nie zostały osiągnięte maksymalne zakresy ruchów w danej osi, pola *rotation* przyjmują wartości *initialLocalRotation - tiltAround*. Oznacza to, że aktualna rotacja obiektu jest modyfikowana o kąt, o jaki ma się obrócić.

Zmienna *target* przyjmuje wartość rotacji rodzica przemnożoną przez kąt, o jaki chcemy, żeby nasz obiekt się obrócił. Gdybyśmy nie uwzględnili rotacji rodzica, część modelu mogłaby się zachowywać w sposób niekontrolowany i przekraczać nałożone na nią ograniczenia.

Ostatecznie modyfikowana jest zmienna *jointTransform* (przechowuje informacje o położeniu i rotacji danego obiektu oraz pozwala na ich manipulację) za pomocą metody *Slerp()*. Metoda ta przemieszcza obiekt z aktualnej pozycji do tej, zapisanej pod zmienną *target*. Możemy zauważyć, że na szybkość z jaką to robi wpływa omawiana wcześniej zmienna *smooth*. Zastosowanie wartości *Time.deltaTime* niejako sprawia, że ruch odbywa się niezależnie od występujących po sobie w danej scenie kratek. Dla wartości *smooth = 10* możemy to rozumieć jako "przemieszczenie z prędkością 10  $\frac{m}{s}$ " zamiast "przemieszczenia z prędkością 10 metrów na kratkę".

### 3.3.3 Schemat komunikacji między ModelController oraz NeuralNetwork

Klasa *ModelController* jest główną klasą sterującą modelem. Odczytuje ona dane o położeniu modelu, które następnie wysyła do sieci neuronowej zaimplementowanej w skrypcie *NeuralNetwork*. Następnie *NeuralNetwork* oblicza odpowiedź, która jest zwracana do *ModelController*, która wykonuje akcje zgodne z otrzymanymi danymi. *ModelController* została napisana w języku C#, natomiast *NeuralNetwork* w języku python. Z tego powodu komunikacja między nimi odbywa się poprzez zapis danych do plików pośrednich - *input* oraz *output*. Wzięły one swoje nazwy odpowiednio od danych wchodzących i wychodzących z sieci neuronowej. Cały schemat komunikacji został przedstawiony na poniższej ilustracji.



Rysunek 3.4: Schemat komunikacji między *ModelController* oraz *NeuralNetwork*.

### 3.3.4 ModelController

```
using UnityEngine;
using System;
using System.IO;

public class ModelController : MonoBehaviour {
    public MoveJoint[] movableParts;
    private Vector3 initialPosition;
    private double[] previousMoves;

    ...
}
```

Klasa *ModelController* oprócz omówionej wcześniej biblioteki *UnityEngine* używa także bibliotek *System* oraz *System.IO*. Pozwalają one na korzystanie z powszechnie stosowanych klas (np. klasa *String*) oraz wykonywanie operacji na plikach tekstowych.

W tablicy *movableParts* przechowywane są wszystkie części modelu, do których został przypisany skrypt *MoveJoint* (tzn. takie, którym możemy sterować).

Pod polem *initialPosition* klasy *Vector3* (służącej w Unity do przechowywania wektorów i punktów w przestrzeni) zapisana zostanie początkowa lokalizacja *Spine*.

Tablica *previousMoves* posłuży do zapisywania poprzedniego położenia każdej części modelu, którą możemy sterować (zostanie do dokładnie omówione podczas opisu metod *SpineForward()* oraz *SpineBackward()*).

```
public static bool IsFileReady(String pathToFile) {
    try {
        using (FileStream inputStream = File.Open(pathToFile,
            FileMode.Open, FileAccess.Read, FileShare.None)) {
            if (inputStream.Length > 0) {
                return true;
            } else {
                return false;
            }
        }
    } catch (Exception) {
        return false;
    }
}
```



Metoda *IsFileReady()* sprawdza, czy w danej chwili plik pod ścieżką przekazaną w argumencie nie jest wykorzystywany przez inny program.

```
public double MeasureDistance() {  
    return gameObject.transform.GetChild(0).GetChild(0).position.x -  
        initialPosition.x;  
}
```

Metoda *MeasureDistance()* mierzy odległość, jaką pokonał *Spine* od miejsca początkowego (mierzymy tylko odległość w wymiarze X).

```
public double SpineForward() {  
    if (previousMoves[0] > -0.9) {  
        previousMoves[0] = previousMoves[0] - 0.1;  
    }  
    return previousMoves[0];  
}  
  
public double SpineBackward() {  
    if (previousMoves[0] < 0.9) {  
        previousMoves[0] = previousMoves[0] + 0.1;  
    }  
    return previousMoves[0];  
}
```

Metody *SpineForward()* i *SpineBackward()* są przykładami metod, których będziemy używać do sterowania modelem. W sumie będziemy się posługiwać dziesięcioma analogicznymi metodami (po dwie przypadające na każdą część modelu, którą możemy poruszać). Każda z nich odczytuje wartość z odpowiedniej komórki tablicy *previousMoves*, a następnie dodaje lub odejmuje od tej wartości 0.1 (w zależności od ruchu, jaki ma uzyskać). Wartość ta musi się mieścić w zakresie od -1.0 do 1.0, ponieważ później będzie przekazywana do metody *Move()* ze skryptu *MoveJoint*. Ostatecznie metoda zapisuje zmienioną wartość w odpowiedniej komórce tablicy.

```
void Start() {  
    initialPosition =  
        gameObject.transform.GetChild(0).GetChild(0).position;  
    MoveJoint[] moveJoints = GetComponentInChildren<MoveJoint>();  
  
    foreach (MoveJoint moveJoint in moveJoints) {  
        moveJoint.Start();  
    }  
  
    movableParts = new MoveJoint[5];  
    movableParts[0] = moveJoints[0]; // Spine  
    movableParts[1] = moveJoints[1]; // UpperLeg.L  
    movableParts[2] = moveJoints[2]; // LowerLeg.L  
    movableParts[3] = moveJoints[3]; // UpperLeg.R  
    movableParts[4] = moveJoints[4]; // LowerLeg.R  
  
    previousMoves = new double[5];  
    for (int i = 0; i < 5; i++) {  
        previousMoves[i] = 0.0;  
    }  
}
```

Metoda *Start()* inicjalizuje wszystkie niezbędne do działania programu zmienne. Zmienna *initialPosition* zapisuje położenie *Spine*. W tablicy *moveJoints* przechowywane są wszystkie obiekty klasy *MoveJoints* występujące w modelu. Następnie każdy z nich jest inicjowany za pomocą metody *Start()* z klasy *MoveJoints*.

Na końcu wszystkie części modelu, którymi możemy sterować, trafiają do tablicy *movableParts*, a wszystkie komórki tablicy *previousMoves* inicjowane są wartością 0.0 (stan początkowy każdej ruchomej części).

```
void Update() {
    double[] inputSignals = new double[11];

    double spinePositionX = movableParts[0].transform.position.x;
    double spinePositionY = movableParts[0].transform.position.y;

    inputSignals[0] = movableParts[0].transform.position.y;
    inputSignals[1] = movableParts[0].transform.rotation.eulerAngles.x;

    for (int i = 1; i < movableParts.Length; i++) {
        inputSignals[(2 * i)] = movableParts[i].transform.position.x -
                                spinePositionX;
        inputSignals[(2 * i) + 1] = movableParts[i].transform.position.y -
                                spinePositionY;
    }
    inputSignals[10] = MeasureDistance();

    String input = "";
    for (int i = 0; i < 11; i++) {
        if (i != 10) {
            input += inputSignals[i] + Environment.NewLine;
        } else {
            input += inputSignals[i];
        }
    }

    bool isFileReady = false;
    while (!isFileReady) {
        isFileReady = IsFileReady(@"path\to\input.txt");
    }
    System.IO.StreamWriter inputFile =
        new System.IO.StreamWriter(@"ptah\to\input.txt");
    inputFile.WriteLine(input);
    inputFile.Close();

    double[] outputSignals = new double[10];

    isFileReady = false;
    while (!isFileReady) {
        isFileReady = IsFileReady(@"path\to\output.txt");
    }
    string outputText = File.ReadAllText(@"path\to\output.txt");
    string[] lines = outputText.Split(new string[] { Environment.NewLine },
                                     StringSplitOptions.None);
    for (int i = 0; i < lines.Length; i++) {
        if (!lines[i].Equals("")) {
            outputSignals[i] = Double.Parse(lines[i]);
        }
    }
}
```



```

double maxOutput = -999999999.0;
int action = -1;
for (int i = 0; i < outputSignals.Length; i++) {
    if (output[i] > maxOutput) {
        maxOutput = outputSignals[i];
        action = i;
    }
}

switch (action) {
    case 0:
        movableParts[0].Move((float)SpineForward(), 0.0F, 0.0F);
        break;
    case 1:
        movableParts[0].Move((float)SpineBackward(), 0.0F, 0.0F);
        break;
    case 2:
        movableParts[1].Move(0.0F, 0.0F, (float)UpperLegLForward());
        break;
    case 3:
        movableParts[1].Move(0.0F, 0.0F, (float)UpperLegLBackward());
        break;
    case 4:
        movableParts[2].Move((float)LowerLegLForward(), 0.0F, 0.0F);
        break;
    case 5:
        movableParts[2].Move((float)LowerLegLBackward(), 0.0F, 0.0F);
        break;
    case 6:
        movableParts[3].Move(0.0F, 0.0F, (float)UpperLegRForward());
        break;
    case 7:
        movableParts[3].Move(0.0F, 0.0F, (float)UpperLegRBackward());
        break;
    case 8:
        movableParts[4].Move((float)LowerLegRForward(), 0.0F, 0.0F);
        break;
    case 9:
        movableParts[4].Move((float)LowerLegRBackward(), 0.0F, 0.0F);
        break;
}
}

```

Metoda *Update()* odpowiada za komunikację ze skrypcem *NeuralNetwork* oraz sterowanie modelem. Jest ona uruchamiana dla każdej nowej kratki w trakcie działania sceny.

Na jej początku tworzona jest tablica *inputSignals* o jedenastu komórkach. Będzie ona przechowywała wszystkie dane o położeniu modelu (jego *stan*). W jej pierwszej komórce zapisywana jest wysokość *Spine* nad podłożem, natomiast w drugiej wychylenie *Spine* od osi pionowej. W komórkach od trzeciej do dziesiątej zapisywane jest położenie części *UpperLeg.L*, *LowerLeg.L*, *UpperLeg.R* oraz *LowerLeg.R* względem *Spine*. W jedenastej komórce zapisywany jest przebyty dystans.

Następnie *Update()* sprawdza w pętli za pomocą metody *IsFileReady()*, czy możliwy jest zapis wartości z *inputSignals* do pliku *input*. Gdy zostaje on zwolniony, dane zostają zapisane.

W dalszej kolejności *Update()* odczytuje odpowiedź sieci neuronowej z pliku *output* i zapisuje ją do tablicy

*outputSignals*. Dalej odszukuje wartość maksymalną w *outputSignals* i zapamiętuje indeks, pod którym się znajdowała.

Na samym końcu, w zależności od indeksu, pod którym znajdowała się wartość maksymalna w sygnale wyjściowym sieci neuronowej, wykonywana jest odpowiadająca mu akcja.

### 3.3.5 NeuralNetwork

```
import numpy as np
import random
import sys, os.path, time, logging
import math
import pickle
# Watchdog – for checking files
from watchdog.observers import Observer
from watchdog.events import PatternMatchingEventHandler
# Keras – for neural network
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import RMSprop

mode = None

if os.path.isfile("path\to\model.p"):
    print("POBIERANIE Z PLIKU...")
    model = pickle.load(open("path\to\model.p", "rb"))
else:
    print("TWORZENIE NOWEGO MODELU")
    model = Sequential()

    model.add(Dense(200, init='lecun_uniform', input_shape=(11,)))
    model.add(Activation('relu'))

    model.add(Dense(150, init='lecun_uniform'))
    model.add(Activation('relu'))

    model.add(Dense(10, init='lecun_uniform'))
    model.add(Activation('linear'))

    rms = RMSprop()
    model.compile(loss='mse', optimizer=rms)
```

Na początku skryptu *NeuralNetwork* pobierane są wszystkie niezbędne biblioteki (omówione w rozdziale 2) i tworzona zmienna *mode*, która będzie mówiła o tym, w jakim trybie będzie działała sieć neuronowa. Mogą to być *learn* - uczenie sieci, oraz *run* - generowanie odpowiedzi. Następnie program sprawdza, czy w danej lokalizacji istnieje plik *model.p*. W pliku tym zapisywana będzie struktura sieci neuronowej. Jest to konieczne, gdy chcemy zakończyć proces uczenia i rozpocząć sterowanie modelem za pomocą sieci. Jeżeli plik taki istnieje, struktura sieci jest z niego wczytywana, a w przeciwnym wypadku tworzona jest zupełnie nowa.

Jeśli nie został wykryty plik z zapisaną siecią, tworzymy obiekt *model* ją reprezentujący (za pomocą konstruktora *Sequential()*). Model *Sequential* definiowany jest jako liniowy stos warstw, do którego za pomocą metody *add(Dense())* dodawane są kolejne warstwy. Do każdej z nich dodawana jest funkcja aktywacji *relu*, którą można zapisać następująco:

$$\varphi(x) = \max\{0, x\}.$$

Powyższy kod tworzy sieć, w której skład wchodzi:



- warstwa wejściowa składającą się z 11 neuronów,
- dwie warstwy ukryte (posiadające odpowiednio 200 i 150 neuronów),
- warstwa wyjściowa złożona z 10 neuronów.

Na samym końcu do modelu dodawana jest funkcja straty (omówiona w podrozdziale 1.4.2) oraz *optimizer*. *Optimizer* ma na celu odpowiednie dobieranie tempa uczenia sieci (siły, z jaką zmieniane będą wagi neuronu). W załączonym przykładzie jako funkcję straty zastosowano *błąd średniokwadratowy* (ang. mean squared error), a za *optimizer* tzw. metodę *RMSPProp*.

```
def getReward(spineInclination, distance, previousDistance, spineHeight):
    if (float(spineInclination) > 2.0 and float(spineInclination) < 30.0) and
        (distance > previousDistance):
        if (float(spineHeight) > 0.8):
            return 10
        else:
            return 5
    elif distance < 0:
        return -5
    else:
        return -1
```

Funkcja *getReward()* zwróci liczbę będącą nagrodą na podstawie argumentów, które otrzymała. Zwróci wartość 5, gdy wychylenie kręgosłupa będzie w zakresie od 2° do 30° i gdy przebyty w danym kroku dystans jest większy, niż w kroku poprzednim. Ponadto, jeśli wysokość *Spine* nad podłożem wynosi 80% wysokości modelu funkcja ta zwróci wartość 10. Jeśli model porusza się do tyłu, *getReward()* zwróci wartość -5. W pozostałych przypadkach nagroda będzie wynosiła -1.

```
def refactorInput(input):
    strength = 0.0
    for i in range(0, 11):
        strength += math.pow(float(input[i]), 2.0)
    strength = math.sqrt(strength)
    for i in range(0, 11):
        input[i] = float(input[i]) / strength
    return input
```

Funkcja *refactorInput()* ma na celu znormalizowanie otrzymanego sygnału wejściowego. Jest to konieczne, ponieważ otrzymane wartości mogą się bardzo od siebie różnić (np. przebyty dystans od wychylenia *Spine* od pionu), co mogłoby negatywnie wpłynąć na proces uczenia.

Na początku obliczana jest siła sygnału wejściowego ze wzoru

$$strength = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2},$$

gdzie  $x_i$  są jego kolejnymi składowymi, a  $n$  reprezentuje ich liczbę.

Następnie każda składowa wektora sygnału wejściowego jest dzielona przez siłę tego sygnału.

```
class Data():
    def __init__(self, previousState, previousOutput, previousAction,
                 previousDistance, epsilon):
        self.previousState = previousState
        self.previousOutput = previousOutput
        self.previousAction = previousAction
        self.previousDistance = previousDistance
        self.epsilon = epsilon
```

Klasa *Data* została stworzona, aby przechowywać wszystkie dane z poprzedniego kroku procesu uczenia sieci. Jej pola odpowiadają następującym informacjom:

- *previousState* - poprzedni sygnał wejściowy sieci neuronowej (stan modelu),
- *previousOutput* - poprzednia odpowiedź sieci,
- *previousAction* - poprzednia podjęta akcja,
- *previousDistance* - dystans, jaki przebył model do poprzedniego kroku,
- *epsilon* - ostatnia wartość  $\epsilon$ .

```
class MyEventHandler(PatternMatchingEventHandler):
    previousDistance = 0.0
    data = Data(previousState=None, previousOutput=None, previousAction=None,
                 previousDistance=previousDistance, epsilon=1.0)

    def on_modified(self, event):
        super(MyEventHandler, self).on_modified(event)
        self.data = runNetwork(previousDistance=self.previousDistance,
                               epsilon=self.data.epsilon, gamma=0.7,
                               previousState=self.data.previousState,
                               previousOutput=self.data.previousOutput,
                               previousAction=self.data.previousAction)
        self.previousDistance = self.data.previousDistance
```

Klasa *MyEventHandler* dziedziczy po klasie *PatternMatchingEventHandler* z pakietu *watchdog*. Jej instancja jest tworzona tylko raz podczas uruchamiania skryptu. Podczas inicjalizacji ustala wartość pola *previousDistance* jako 0.0 (ponieważ przy uruchamianiu sceny model nie przebył jeszcze żadnej odległości). Następnie tworzy obiekt klasy *Data* z pustymi polami (z wyjątkiem *epsilon*, który wynosi 1.0), w którym będą zapamiętywane dane z modelu i sieci z poprzedniego kroku działania programu. Zdefiniowana została w niej metoda *on\_modified()*, która jest uruchamiana za każdym razem, gdy w monitorowanym pliku zostały wprowadzone zmiany. Pozwala na to metoda *on\_modified* z klasy nadrzędnej *PatternMatchingEventHandler*. Po każdej wprowadzonej w pliku (*input*) zmianie, za pomocą metody *runNetwork()* zostaje uruchomiona sieć neuronowa, która generuje odpowiedź i jednocześnie zapisuje ją wraz z danymi o modelu w *data*. Na końcu zostaje zaktualizowana wartość *previousDistance*.

```
def runNetwork(previousDistance, epsilon, gamma, previousState, previousOutput,
               previousAction):
    while True:
        try:
            with open("path\to\input.txt", "r") as file:
                input = file.readlines()
                input = [x.strip('\n') for x in input]
                break
        except IOError:
            pass
    data = Data(previousState=None, previousOutput=None, previousAction=None,
                 previousDistance=None, epsilon=epsilon)
    y = [[previousOutput]]

    if input.__len__() == 11:
        if previousState is None or previousOutput is None or
            previousAction is None:
            previousState = input
            previousOutput = model.predict(np.reshape(input, (1, 11)),
```





```

                                batch_size=1)
    previousAction = 0
    spineHeight = input[0]
    spineInclination = input[1]
    distance = input[10]
    if mode == "learn":
        reward = getReward(spineInclination, distance, previousDistance,
                            spineHeight)
        input = refactorInput(input)
        newOutput = model.predict(np.reshape(input, (1, 11)), batch_size=1)
        maxOutput = np.max(newOutput)
        y = np.zeros((1, 10))
        y[:] = previousOutput[:]
        if reward == -1:
            update = (reward + (gamma * maxOutput))
        else:
            update = reward
        y[0][previousAction] = update
        model.fit(np.reshape(previousState, (1, 11)), y, batch_size=1,
                  nb_epoch=1, verbose=1)
        output = model.predict(np.reshape(input, (1, 11)), batch_size=1)
        y[:] = output[:]
        if (random.random() < epsilon):
            action = np.random.randint(0, 10)
        else:
            action = (np.argmax(output))
        if epsilon > 0.1:
            epsilon -= 0.00005
    else:
        input = refactorInput(input)
        output = model.predict(np.reshape(input, (1, 11)), batch_size=1)
        y[:] = output[:]
        action = (np.argmax(output))
    data.previousState = input
    data.previousOutput = output
    data.previousAction = action
    data.previousDistance = distance
    data.epsilon = epsilon
    outputStr = ""
    for i in range(len(y[0])):
        outputStr += str(y[0][i]) + '\n'
    while True:
        try:
            with open("path\to\output.txt", 'w') as file:
                file.write(outputStr)
                file.close()
            break
        except IOError:
            pass
    return data

```

Metoda *runNetwork()* jest główną metodą odpowiedzialną za uczenie się sieci oraz generowanie jej odpowiedzi. Na początku w pętli sprawdza ona, czy został już zwolniony plik *input*. Gdy możliwy jest już jego odczyt, metoda pobiera dane wejściowe sieci i zapisuje je pod zmienną *input*. Następnie tworzony jest

pusty obiekt *data*, a pod zmienną *y* podstawiana jest poprzednia odpowiedź sieci (w formacie czytelnym dla metody *fit()* pakietu *Keras*). Następnie *runNetwork()* sprawdza, czy odczytane dane wejściowe są pożądanej długości (tzn. czy zostało przekazane 11 wartości). Jeśli warunek ten nie został spełniony, sieć nie będzie obliczać nowej odpowiedzi, a zwróci tę z poprzedniej iteracji. Z tego powodu zmienne *data* oraz *y* zostały zainicjowane przed warunkiem "if *input.\_\_len\_\_()* == 11:".

Jeśli wektor sygnału wejściowego spełnił warunek wejściowy, sprawdzane jest, czy zmienne *previousState*, *previousOutput* oraz *previousAction* mają wartość różną od *None*. Jedynym przypadkiem, gdy mogą posiadać taką wartość, jest pierwszy krok działania programu. Przyjmują one wtedy następujące dane:

- *previousState* - aktualny stan modelu,
- *previousOutput* - obliczoną nową odpowiedź sieci,
- *previousAction* - akcję 0 ( $a_0$ ).

W dalszej części metody *runNetwork()* z sygnału wejściowego odczytywane są dane o wysokości *Spine* nad podłożem, jego wychylenie od pionu oraz przebyty dystans.

Jeśli sieć działa w trybie uczenia się, na podstawie odczytanych danych obliczana jest nagroda, która jest zapisywana pod zmienną *reward*. Następnie sygnał wejściowy zostaje znormalizowany metodą *refactorInput()* i obliczana jest na jego podstawie odpowiedź sieci. Z odpowiedzi wydobywana jest wartość maksymalna i zapisywana pod zmienną *maxOutput*.

Pod zmienną *y* podstawiana jest odpowiedź sieci z poprzedniej iteracji. Jeśli wyliczona nagroda jest równa -1, zmienna *update* przyjmuje wartość  $reward + (gamma * maxOutput)$ . W przeciwnym wypadku (dla nagród 10, 5 i -5) *update* jest równa uzyskanej nagrodzie.

Pod miejsce w tablicy *y* odpowiadające wykonanej uprzednio akcji podstawiana jest wartość *update*. Na podstawie tak zmodyfikowanego sygnału wyjściowego sieć modyfikuje swoje wagi za pomocą *model.fit()* (zgodnie z zasadą propagacji wstecznej). Następnie, poprzez wywołanie *model.predict()*, sieć generuje odpowiedź dla aktualnego stanu modelu, zapisuje ją pod zmienną *output* i kopiuje ją do zmiennej *y* (która zostanie zapisana do pliku *output*).

W dalszej części generowana jest losowa liczba z przedziału od 0.0 do 1.0. Jeśli jest ona mniejsza od aktualnej wartości *epsilon*, zostaje wybrana losowa akcja. Jeśli natomiast uzyskana liczba jest większa od *epsilon*, wykonywana jest akcja wskazana przez *output*.

Na końcu, jeśli wartość *epsilon* jest większa od 0.1, zmniejszana jest ona o 0.00005.

Jeśli natomiast sieć działa w trybie generowania odpowiedzi, po odczytaniu danych o wysokości *Spine* nad podłożem, jego wychyleniu od pionu oraz przebytym dystansie, sygnał wejściowy *input* jest normalizowany metodą *refactorInput()* i na jego podstawie jest generowana odpowiedź sieci (zapisywana pod zmienną *output*). Następnie na podstawie *output* wybierana jest odpowiednia akcja.

Po wykonaniu powyższych instrukcji *runNetwork()* przetwarza odpowiedź sieci na formę możliwą do zapisu w pliku. Gdy plik *output* zostaje zwolniony, sygnał wyjściowy zostaje w nim zapisany.



```
def main():
    file_path = "path\to\input.txt".strip()
    watched_dir = os.path.split(file_path)[0]
    patterns = [file_path]
    event_handler = MyEventHandler(patterns=patterns)
    observer = Observer()
    observer.schedule(event_handler, watched_dir, recursive=True)
    observer.start()
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
        pickle.dump(model, open("path\to\model.p", "wb"))
    observer.join()

if __name__ == "__main__":
    if len(sys.argv) > 1:
        mode = sys.argv[1]
        if mode != "run" and mode != "learn":
            print("Podano niepoprawny tryb działania sieci")
            sys.exit(1)
        else:
            main()
    else:
        sys.exit(1)
```

W metodzie *main()* zmienna *file\_path* przechowuje adres pliku *input*. Za pomocą metody *strip()* zostaje ona pozbawiona z początku i końca domyślnych białych znaków. Następnie, pod zmienną *watched\_dir*, zapisana zostaje *głowa* adresu pliku. Jest to cały adres z wyjątkiem jego ostatniej części (po znaku "\"). W dalszej części tworzony jest obiekt klasy *MyEventHandler* i obserwator pliku *input*.

Metoda *main()* w pętli nasłuchuje, czy użytkownik nie zakończył ręcznie działania programu. Jest to jedyny sposób na przerwanie działania sieci neuronowej i zapisania struktury sieci neuronowej do pliku *model.p*. Ponadto, metoda ta sprawdza, czy program został uruchomiony z poprawnym argumentem (mówiącym o tym, w jakim trybie ma działać sieć). Jeśli jest on różny od "run" i "learn" lub nie został w ogóle przekazany, program kończy działanie. W przeciwnym wypadku zostaje on przypisany do zmiennej *mode*.

# Wyniki przeprowadzonych testów oraz płynące z nich wnioski

W rozdziale 3 omówiliśmy dokładnie działanie całego programu. Możemy zauważyć, że na jego wynik końcowy wpływ mają następujące czynniki:

- dobrana funkcja nagrody,
- wartość *gamma*,
- szybkość spadku wartości *epsilon*,
- struktura sieci neuronowej.

W trakcie przeprowadzania testów modyfikowane były jedynie wartości *gamma*, długość procesu uczenia (poprzez zmianę szybkości spadku wartości *epsilon* w każdym kroku procesu uczenia sieci) oraz struktura sieci neuronowej (rozpatrywane były jedynie sieci z jedną i dwoma warstwami ukrytymi). Przyjęta funkcja nagrody pozostała bez zmian. Miara skuteczności testowanego przypadku był dystans, jaki model przebył od miejsca początkowego.

W załączonych tabelach struktury sieci będziemy oznaczać następująco:

$$X \times Y \times Z.$$

Oznaczać to będzie, że sieć ta ma w warstwie wejściowej *X* neuronów, *Y* w ukrytej i *Z* w wyjściowej. Dla sieci z dwoma warstwami ukrytymi oznaczenie to będzie wyglądało analogicznie:

$$X \times Y_1 \times Y_2 \times Z.$$

Proces uczenia sieci dzielimy na kroki. Ich liczba (ozn. przez *t*) obliczana jest następująco:

$$t = \frac{0.9}{\epsilon}.$$

$\epsilon$  oznacza wartość, o jaką będzie się w każdym kroku zmniejszał *epsilon*. W omawianym skrypcie *Neural-Network* (3.3.5) reprezentowała to linia "*epsilon -= 0.00005*". Licznik w powyższym ułamku równy jest 0.9, ponieważ na samym początku procesu uczenia sieci *epsilon* wynosi 1.0, a gdy proces ten jest przerywany, jego wartość spada do 0.1.

Przebyty dystans będzie obliczany za pomocą metody *MeasureDistance()*, przedstawionej w podrozdziale 3.3.4.



## 4.1 Wyniki testów dla sieci z jedną warstwą ukrytą

Gamma	Liczba kroków w procesie uczenia sieci	Struktura sieci	Przebyty dystans	Wykonywane akcje
0.3	9000	11 x 200 x 10	0.32	6
0.5	18000	11 x 500 x 10	0.26	1, 2, 3, 4, 6
0.9	90000	11 x 1000 x 10	0.39	2, 6, 8

Tablica 4.1: Wyniki testów dla trzech losowych przypadków.

Dla pierwszych trzech badanych przypadków możemy zauważyć, że zwiększenie liczby neuronów w warstwie ukrytej wpłynęło na zróżnicowanie akcji, które wykonywał model. Ponadto zbytne zwiększenie liczby kroków w procesie uczenia, nawet dla sieci zawierającej 1000 neuronów w warstwie ukrytej, doprowadziło do tzw. *przeuczenia* (ang. *overfitting*). Mimo potencjalnie większych możliwości, sieć ta zlecała wykonanie mniejszej liczby akcji, niż sieć posiadająca 500 neuronów w warstwie ukrytej.

W poniższej tabeli przedstawiono wyniki dla zmieniającej się liczby kroków w procesie uczenia się sieci i stałej wartości *gamma* oraz struktury sieci.

Gamma	Liczba kroków w procesie uczenia sieci	Struktura sieci	Przebyty dystans	Wykonywane akcje
0.3	9000	11 x 250 x 10	0.36	2
0.3	18000	11 x 250 x 10	0.07	0, 3
0.3	30000	11 x 250 x 10	0.29	5, 6, 8, 9

Tablica 4.2: Wyniki dla zmieniającej się liczby kroków w procesie uczenia się sieci.

Zauważmy, że po zakończeniu procesu uczenia się, który trwał 9000 kroków, sieć wskazuje tylko jedną akcję do wykonania. Pokrywa się to z wynikami przedstawionymi w tabeli 4.1. Jest to zatem zbyt krótki czas, by sieć mogła nauczyć się podejmowania bardziej skomplikowanych decyzji.

W poniższych tabelach przedstawiono wyniki dla dwóch różnych struktur sieci, w których proces uczenia się wynosił 18 000 lub 30 000 kroków. Zmieniała się także wartości *gamma*. Najniższa liczba neuronów w warstwie ukrytej wynosiła 500, ponieważ dla takiego przypadku rezultaty przy 18 000 krokach w procesie uczenia się sieci były bardziej zróżnicowane, niż dla 250 neuronów (patrz tabela 4.1).

Gamma	Liczba kroków w procesie uczenia sieci	Struktura sieci	Przebyty dystans	Wykonywane akcje
0.3	18000	11 x 500 x 10	0.42	4, 5, 8
0.3	30000	11 x 500 x 10	0.77	0, 5
0.6	18000	11 x 500 x 10	0.05	0
0.6	30000	11 x 500 x 10	0.22	0, 7
0.9	18000	11 x 500 x 10	0.36	2
0.9	30000	11 x 500 x 10	0.03	0

Tablica 4.3: Wyniki testów dla sieci, w której warstwie ukrytej było 500 neuronów.

Gamma	Liczba kroków w procesie uczenia sieci	Struktura sieci	Przebyty dystans	Wykonywane akcje
0.3	18000	11 x 1000 x 10	0.32	6
0.3	30000	11 x 1000 x 10	0.76	0, 5
0.6	18000	11 x 1000 x 10	0.24	0, 3
0.6	30000	11 x 1000 x 10	0.05	0
0.9	18000	11 x 1000 x 10	0.01	0
0.9	30000	11 x 1000 x 10	0.70	0, 5

Tablica 4.4: Wyniki testów dla sieci, w której warstwie ukrytej było 1000 neuronów.

Wyniki przeprowadzonych testów pokazują, że nawet dla bardzo zbliżonych warunków początkowych, uzyskiwane zostawały różne wyniki. Obrazuje nam to jak duży wpływ na końcowe działanie sieci mają początkowe, losowe wartości wag neuronów. Ponadto, zwiększenie czasu nauczania sieci z reguły zwiększało przebyty przez model dystans.

## 4.2 Wyniki testów dla sieci z dwoma warstwami ukrytymi

W poniższej tabeli zamieszczono wyniki dla początkowo losowo dobranych wartości *gamma*, długości procesu uczenia i struktur sieci neuronowej. Maksymalna liczba neuronów w każdej warstwie wynosiła 750, ponieważ obliczenia przy w bardziej rozbudowanej strukturze sieci trwały zbyt długo, przez co sieć nie dawała natychmiastowych sygnałów sterujących do modelu.

Gamma	Liczba kroków w procesie uczenia sieci	Struktura sieci	Przebyty dystans	Wykonywane akcje
0.3	18000	11 x 200 x 150 x 10	0.42	1, 4
0.45	9000	11 x 150 x 90 x 10	0.26	5, 6
0.6	30000	11 x 500 x 500 x 10	0.92	0, 5, 6, 9
0.75	30000	11 x 750 x 750 x 10	0.22	0
0.9	9000	11 x 50 x 25 x 10	0.34	2

Tablica 4.5: Wyniki testów dla pięciu losowych przypadków.

Podobnie jak w poprzednim podrozdziale możemy zauważyć, że sieci z małą liczbą neuronów w każdej warstwie generują akcje o małej różnorodności.

W następnej tabeli przedstawiono wyniki dla sieci posiadającej 500 neuronów w każdej warstwie ukrytej, której długość procesu uczenia wynosiła 18 000 lub 30 000 kroków, a *gamma* przyjmowała wartości 0.3, 0.45 oraz 0.6.



Gamma	Liczba kroków w procesie uczenia sieci	Struktura sieci	Przebyty dystans	Wykonywane akcje
0.3	18000	11 x 500 x 500 x 10	0.46	8
0.3	30000	11 x 500 x 500 x 10	0.32	6
0.45	18000	11 x 500 x 500 x 10	0.73	6, 7, 9
0.45	30000	11 x 500 x 500 x 10	0.04	0
0.6	18000	11 x 500 x 500 x 10	0.42	0, 1, 7, 8
0.6	30000	11 x 500 x 500 x 10	0.03	0

Tablica 4.6: Wyniki testów dla sieci, w której warstwach ukrytych było po 500 neuronów.

Najlepsze rezultaty odniosły sieci, który proces uczenia się trwał 18 000 kroków, a wartość *gamma* wynosiła 0.45 lub 0.6.

Po porównaniu wyników z dwóch powyższych tabel możemy zauważyć, że dla sieci posiadającej 500 neuronów w każdej warstwie ukrytej, *gamma* równej 0.6 oraz 30 000 kroków procesu uczenia otrzymaliśmy bardzo zróżnicowane wyniki. Zjawisko to potwierdza założenie z poprzedniej sekcji, mówiące o tym, że końcowe działanie programu w dużej mierze zależy od początkowych wartości wag neuronów.

Poniższe tabele przedstawiają wyniki dla dwóch różnych struktur sieci z takimi samymi wartościami *gamma* oraz liczbą kroków procesu nauczania jak w tabeli 4.6.

Gamma	Liczba kroków w procesie uczenia sieci	Struktura sieci	Przebyty dystans	Wykonywane akcje
0.3	18000	11 x 300 x 250 x 10	0.62	0, 2, 8
0.3	30000	11 x 300 x 250 x 10	0.26	0, 2, 5, 7
0.45	18000	11 x 300 x 250 x 10	0.34	5
0.45	30000	11 x 300 x 250 x 10	0.73	0, 4
0.6	18000	11 x 300 x 250 x 10	0.51	4, 8
0.6	30000	11 x 300 x 250 x 10	0.14	0, 8

Tablica 4.7: Wyniki testów dla sieci, w której warstwach ukrytych było 300 oraz 250 neuronów.

Gamma	Liczba kroków w procesie uczenia sieci	Struktura sieci	Przebyty dystans	Wykonywane akcje
0.3	18000	11 x 600 x 400 x 10	0.28	6, 9
0.3	30000	11 x 600 x 400 x 10	0.19	0, 1
0.45	18000	11 x 600 x 400 x 10	0.21	1, 4, 6, 7
0.45	30000	11 x 600 x 400 x 10	0.02	1, 3
0.6	18000	11 x 600 x 400 x 10	0.45	6, 8
0.6	30000	11 x 600 x 400 x 10	0.71	0, 4

Tablica 4.8: Wyniki testów dla sieci, w której warstwach ukrytych było 600 oraz 400 neuronów.

### 4.3 Wnioski

Przedstawione wyniki pokazują, w jak dużym stopniu końcowe działanie sieci neuronowej zależy od dobieranych wartości *gamma* (siły, z jaką zmieniane są wagi neuronów), długości procesu nauczania oraz wybranej struktury sieci. Sieć, o zbyt prostej strukturze generowała sygnały sterujące dużo mniej zróżnicowane, niż te, zwracane przez sieci o większej liczbie neuronów w warstwach ukrytych. Jednakże nawet skomplikowane sieci przy zbyt długim czasie uczenia się ulegały zjawisku *przeuczenia*, co negatywnie wpływało na otrzymywane rezultaty. Zjawisko to mogło potęgować zbyt duże zwiększenie wartości *gamma*.

Wartym zauważenia jest również fakt, że dla rozpatrywanych identycznych przypadków testowych otrzymywane były różne rezultaty. Obrazuje to wagę, jaką mają początkowe "skłonności" neuronów (losowe wartości wag) na końcowe działanie sieci.

W przedstawionym podejściu posługiwaliśmy się bardzo ograniczonymi zbiorami *stanów* modelu oraz możliwych do podjęcia *akcji*. Ponadto, badaliśmy jedynie zachowanie sieci o maksymalnie dwóch warstwach ukrytych. Możliwym jest, że przy zwiększeniu liczby wykrywanych *stanów* modelu (poprzez odczytywanie dodatkowych danych o jego położeniu) i *akcji*, udoskonaleniu *funkcji nagrody* oraz zastosowaniu sieci o dużo bardziej zróżnicowanej strukturze (np. z większą liczbą warstw ukrytych), osiągnięte wyniki byłyby dużo lepsze od przedstawionych w niniejszej pracy.



## Książki

- R. Tadeusiewicz, T. Gąciarz, B. Borowik, B. Leper - "Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#"

## Źródła internetowe

- <https://en.wikipedia.org/wiki/Backpropagation>
- [http://osilek.mimuw.edu.pl/index.php?title=Sztuczna\\_inteligencja/SI\\_Modu%C5%82'13'-Uczenie\\_si%C4%99'ze\\_wzmocnieniem](http://osilek.mimuw.edu.pl/index.php?title=Sztuczna_inteligencja/SI_Modu%C5%82'13'-Uczenie_si%C4%99'ze_wzmocnieniem)