

---

# Spis treści

0.1.	Mechanika Lagrange'a . . . . .	2
0.1.1.	Zagadnienia teoretyczne . . . . .	2
0.1.2.	Dowód równania Eulera-Lagrange'a . . . . .	3
0.1.3.	Dodatkowe terminy . . . . .	4
0.1.4.	Opis problemu . . . . .	5
0.1.5.	Zadania . . . . .	9
0.2.	Sieci neuronowe . . . . .	9
0.2.1.	Architektura sieci . . . . .	10
0.2.2.	Funkcja aktywacji . . . . .	13
0.2.3.	Funkcja kosztu . . . . .	16
0.2.4.	Algorytmy optymalizacyjne . . . . .	17
0.2.5.	Dodatkowe terminy . . . . .	18
0.3.	Rekurencyjne sieci neuronowe . . . . .	18
0.3.1.	Wprowadzenie . . . . .	18
0.3.2.	Architektura . . . . .	19
0.3.3.	Pamień komórek . . . . .	21
0.3.4.	LSTM . . . . .	21
0.4.	Lagranżowskie sieci neuronowe . . . . .	24
0.5.	Badania . . . . .	25
0.5.1.	Uzyskanie danych za pomocą metod numerycznych . . . . .	25
0.5.2.	Uzyskanie danych za pomocą estymacji pozycji rzeczywistego wahadła podwójnego . . . . .	27
0.5.3.	Model przy użyciu RNN oparty na danych uzyskanych numerycznie, nauczony na współrzędnych . . . . .	33
0.5.4.	Model przy użyciu RNN oparty na danych uzyskanych numerycznie, nauczony na współrzędnych i prędkościach . . . . .	35
0.5.5.	Model przy użyciu LNN oparty na danych uzyskanych numerycznie . . . . .	36
<b>Bibliografia</b>	. . . . .	39

Spis rysunków . . . . .	41
-------------------------	----

Aleksander Kaluta

Lagranżowskie sieci neuronowe

## 0.1. Mechanika Lagrange'a

### 0.1.1. Zagadnienia teoretyczne

[4] W tym rozdziale skupimy się na równaniu ruchu, które zostało opracowane przez Eulera i rozbudowane przez Lagrange'a, a jego podstawą jest tzw. zasada akcji stacjonarnej. Kluczowym pytaniem, które sobie zadajemy, jest: jeśli znamy stan układu na początku i na końcu ruchu, to po jakiej trasie (trajektorii) poruszy się cząstka, przemieszczając się między tymi punktami? Ta „zasada” jest ważna, ponieważ choć dokładnie nie wiadomo, dlaczego akurat to podejście prowadzi do poprawnych równań ruchu, to na nim bazuje wiele modeli fizycznych.

Aby to prześledzić, zaczniemy od wprowadzenia dwóch kluczowych wielkości związanych z prędkością, które pomogą nam w wyznaczeniu równań ruchu. Jedną z tych wielkości jest funkcja zwana Lagrangianem, zapisywana jako  $L = L(q, \dot{q}, t)$ . Lagrangian opisuje energię układu i pozwala nam wyprowadzić poprawne równania ruchu. Może się zdarzyć, że dla jednego zjawiska istnieje więcej niż jeden Lagrangian, bo różne podejścia mogą pokazywać ten sam efekt.

Najprostszy sposób zapisania Lagrangianu to różnica między energią kinetyczną a potencjalną:

$$L(q, \dot{q}, t) = T(\dot{q}) - U(q) \quad (1)$$

gdzie:

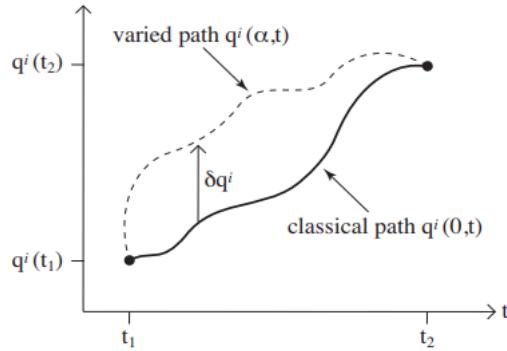
- $T(\dot{q})$  to energia kinetyczna, zależna od prędkości (czyli tempa zmiany położenia),
- $U(q)$  to energia potencjalna, zależna od położenia układu.

Dla niektórych systemów Lagrangian może też zależeć od czasu – wtedy analizujemy to zależnie od kontekstu.

[3] Drugą wielkością jest zasada rzążąca ruchem układów mechanicznych, czyli zasada najmniejszej akcji, znana również jako zasada Hamiltona. Zgodnie z nią każdy układ mechaniczny można opisać za pomocą konkretnej funkcji, którą zapisujemy jako  $L(q_1, \dots, q_n, \dot{q}_1, \dots, \dot{q}_n, t)$  lub w skrócie jako  $L(q, \dot{q}, t)$ . Ruch tego układu odbywa się zgodnie z pewnym warunkiem. Rozważmy sytuację, w której układ znajduje się w chwilach  $t_1$  i  $t_2$  na pozycjach określonych przez dwa różne zestawy współrzędnych:  $q(t_1)$  oraz  $q(t_2)$ . Zasada ta mówi, że układ przemieszcza się między tymi pozycjami w taki sposób, aby całka

$$S = \int_{t_1}^{t_2} L(q, \dot{q}, t) dt \quad (2)$$

osiągała najmniejszą możliwą wartość.



Rysunek 1: Rachunek wariacyjny.

### 0.1.2. Dowód równania Eulera-Lagrange'a

[3] Przejdźmy do wyprowadzenia równań różniczkowych, które rozwiązują problem minimalizacji całki (2). Na początku założymy, że układ ma tylko jeden stopień swobody, co oznacza, że musimy określić jedną funkcję  $q(t)$ .

Niech  $q = q(t)$  będzie funkcją, dla której całka  $S$  osiąga minimum. To oznacza, że  $S$  wzrasta, jeśli  $q(t)$  zastąpimy inną funkcją w formie

$$q(t) + \delta q(t), \quad (3)$$

gdzie  $\delta q(t)$  jest małą funkcją w przedziale czasowym od  $t_1$  do  $t_2$ . Funkcję  $\delta q(t)$  nazywamy wariacją funkcji  $q(t)$ . Ponieważ dla  $t = t_1$  i  $t = t_2$  wszystkie funkcje muszą przyjmować wartości  $q(t_1)$  i  $q(t_2)$  odpowiednio, mamy stąd

$$\delta q(t_1) = \delta q(t_2) = 0.$$

Zmiana całki  $S$  po zastąpieniu  $q$  przez  $q + \delta q$  wynosi

$$S = \int_{t_1}^{t_2} L(q + \delta q, \dot{q} + \delta \dot{q}, t) dt - \int_{t_1}^{t_2} L(q, \dot{q}, t) dt.$$

Gdy rozwiniemy tę różnicę w potęgach  $\delta q$  i  $\delta \dot{q}$ , dominujące składniki będą pierwszego rzędu. Aby  $S$  miało minimum, te składniki (nazywane pierwszą wariacją lub po prostu wariacją całki) muszą być równe zeru. Zatem zasada najmniejszej akcji zapisuje się jako

$$\delta S = \int_{t_1}^{t_2} \left( \frac{\partial L}{\partial q} \delta q + \frac{\partial L}{\partial \dot{q}} \delta \dot{q} \right) dt = 0,$$

lub, po wykonaniu wariacji,

$$\int_{t_1}^{t_2} \left( \frac{\partial L}{\partial q} \delta q + \frac{\partial L}{\partial \dot{q}} \frac{d(\delta q)}{dt} \right) dt = 0.$$

Ponieważ  $\delta \dot{q} = \frac{d(\delta q)}{dt}$ , stosując całkowanie przez części w drugim składniku, otrzymujemy

$$\int_{t_1}^{t_2} \frac{\partial L}{\partial \dot{q}} \frac{d(\delta q)}{dt} dt = \left[ \frac{\partial L}{\partial \dot{q}} \delta q \right]_{t_1}^{t_2} - \int_{t_1}^{t_2} \frac{d}{dt} \left( \frac{\partial L}{\partial \dot{q}} \right) \delta q dt.$$

Warunki  $\delta q(t_1) = \delta q(t_2) = 0$  oznaczają, że zintegrowany składnik w powyższym równaniu jest równy零. Pozostaje więc całka, która musi zniknąć dla wszystkich wartości  $\delta q$ . Może tak być tylko wtedy, gdy całkowany składnik jest identycznie równy零. W ten sposób uzyskujemy

$$\frac{d}{dt} \left( \frac{\partial L}{\partial \dot{q}} \right) = \frac{\partial L}{\partial q}.$$

Gdy układ ma więcej niż jeden stopień swobody, różne funkcje  $q_i(t)$  muszą być wariowane niezależnie w ramach zasady najmniejszej akcji. Otrzymujemy wówczas  $s$  równań w postaci

$$\frac{d}{dt} \left( \frac{\partial L}{\partial \dot{q}_i} \right) = \frac{\partial L}{\partial q_i}.$$

To są wymagane równania różniczkowe, znane w mechanice jako równania Lagrange'a. Kiedy Lagrangian układu mechanicznego jest znany, równania (2.6) dostarczają zależności między przyspieszeniami, prędkościami i współrzędnymi, czyli są równaniami ruchu tego układu.

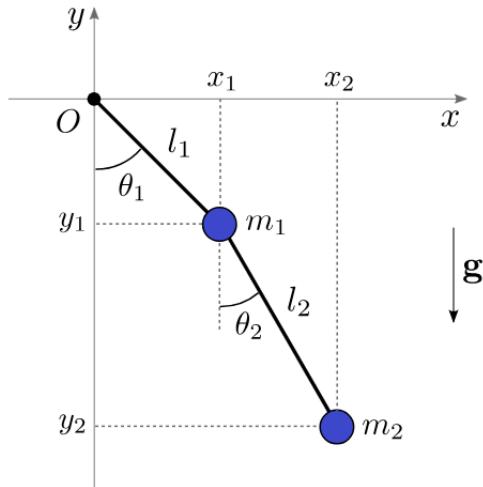
### 0.1.3. Dodatkowe terminy

stopień swobody

### 0.1.4. Opis problemu

Równania Lagrange'a możemy wykorzystać do rozwiązania naszego problemu. Chcemy stworzyć model ruchu wahadła podwójnego. Do tego celu będziemy potrzebowali obliczyć dynamikę modelu, a następnie za pomocą metod numerycznych otrzymać ruch wahadła.

Na początku opiszmy model wahadła podwójnego



$$x_1 = l_1 \sin \theta_1 \quad (4)$$

$$x_2 = l_1 \sin \theta_1 + l_2 \sin \theta_2 \quad (5)$$

$$y_1 = -l_1 \cos \theta_1 \quad (6)$$

$$y_2 = -l_1 \cos \theta_1 - l_2 \cos \theta_2 \quad (7)$$

Następnie zróżniczkujmy te zmienne, aby policzyć prędkości:

$$\dot{x}_1 = l_1 \dot{\theta}_1 \cos \theta_1 \quad (8)$$

$$\dot{x}_2 = l_1 \dot{\theta}_1 \cos \theta_1 + l_2 \dot{\theta}_2 \cos \theta_2 \quad (9)$$

$$\dot{y}_1 = l_1 \dot{\theta}_1 \sin \theta_1 \quad (10)$$

$$\dot{y}_2 = l_1 \dot{\theta}_1 \sin \theta_1 + l_2 \dot{\theta}_2 \sin \theta_2 \quad (11)$$

Policzmy teraz energię kinetyczną T i potencjalną V obiektu, aby otrzymać Lagranżian L.

$$T = \frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 = \frac{1}{2}m_1(\dot{x}_1^2 + \dot{y}_1^2) + \frac{1}{2}m_2(\dot{x}_2^2 + \dot{y}_2^2) = \quad (12)$$

$$= \frac{1}{2}m_1l_1^2\dot{\theta}_1^2(\sin^2\theta + \cos^2\theta) + \frac{1}{2}m_2l_1^2\dot{\theta}_1^2(\sin^2\theta + \cos^2\theta) + \frac{1}{2}m_2l_2^2\dot{\theta}_2^2(\sin^2\theta + \cos^2\theta) + \\ + m_2l_1l_2\dot{\theta}_1\dot{\theta}_2(\cos\theta_1\cos\theta_2 + \sin\theta_1\sin\theta_2) = \quad (13)$$

$$+ m_2l_1l_2\dot{\theta}_1\dot{\theta}_2(\cos\theta_1\cos\theta_2 + \sin\theta_1\sin\theta_2) = \quad (14)$$

$$= \frac{1}{2}m_1l_1^2\dot{\theta}_1^2 + \frac{1}{2}m_2[l_1^2\dot{\theta}_1^2 + l_2^2\dot{\theta}_2^2 + 2l_1l_2\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2)] \quad (15)$$

$$V = m_1gy_1 + m_2gy_2 = m_1gl_1\cos\theta_1 - m_2g(l_1\cos\theta_1 + l_2\cos\theta_2) = \quad (16)$$

$$= -(m_1 + m_2)gl_1\cos\theta_1 - m_2gl_2\cos\theta_2 \quad (17)$$

$$L = T - V = \frac{1}{2}(m_1 + m_2)l_1^2\dot{\theta}_1^2 + \frac{1}{2}m_2l_2^2\dot{\theta}_2^2 + m_2l_1l_2\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2) + \quad (18)$$

$$+ (m_1 + m_2)gl_1\cos\theta_1 + m_2gl_2\cos\theta_2 \quad (19)$$

Skoro mamy już Lagranzian, teraz podstawimy go do wzoru Eulera-Lagrange'a względem parametrów  $\theta_1$  and  $\theta_2$ , aby otrzymać zachodzącą dynamikę w układzie.

$$\frac{d}{dt}\left(\frac{\partial L}{\partial\dot{\theta}_i}\right) - \frac{\partial L}{\partial\theta_i} = 0 \implies \frac{dp_{\theta_i}}{dt} - \frac{\partial L}{\partial\theta_i} = 0 \quad (20)$$

dla  $i = 1, 2$ . Gdzie:

$$p_{\theta_1} = \frac{\partial L}{\partial\dot{\theta}_1} = (m_1 + m_2)l_1^2\dot{\theta}_1 + m_2l_1l_2\dot{\theta}_2\cos(\theta_1 - \theta_2) \quad (21)$$

$$p_{\theta_2} = \frac{\partial L}{\partial\dot{\theta}_2} = m_2l_2^2\dot{\theta}_2 + m_2l_1l_2\dot{\theta}_1\cos(\theta_1 - \theta_2) \quad (22)$$

$$\frac{dp_{\theta_1}}{dt} = (m_1 + m_2)l_1^2\ddot{\theta}_1 + m_2l_1l_2\ddot{\theta}_2\cos(\theta_1 - \theta_2) \quad (23)$$

$$- m_2l_1l_2\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) + m_2l_1l_2\dot{\theta}_2^2\sin(\theta_1 - \theta_2) \quad (24)$$

$$\frac{dp_{\theta_2}}{dt} = m_2l_2^2\ddot{\theta}_2 + m_2l_1l_2\ddot{\theta}_1\cos(\theta_1 - \theta_2) \quad (25)$$

$$- m_2l_1l_2\dot{\theta}_1^2\sin(\theta_1 - \theta_2) + m_2l_1l_2\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) \quad (26)$$

$$\frac{\partial L}{\partial\theta_1} = -m_2l_1l_2\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) - (m_1 + m_2)gl_1\sin\theta_1 \quad (27)$$

$$\frac{\partial L}{\partial\theta_2} = m_2l_1l_2\dot{\theta}_1\dot{\theta}_2\sin(\theta_1 - \theta_2) - m_2gl_2\sin\theta_2 \quad (28)$$

$$(29)$$

W taki sposób otrzymujemy 2 równania:

$$(m_1 + m_2)l_1^2 \ddot{\theta}_1 + m_2 l_1 l_2 \ddot{\theta}_2 \cos(\theta_1 - \theta_2) - m_2 l_1 l_2 \dot{\theta}_2^2 \sin(\theta_1 - \theta_2) + \quad (30)$$

$$-m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \sin(\theta_1 - \theta_2) + m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \sin(\theta_1 - \theta_2) + (m_1 + m_2)g l_1 \sin \theta_1 = 0 \quad (31)$$

$$m_2 l_2^2 \ddot{\theta}_2 + m_2 l_1 l_2 \ddot{\theta}_1 \cos(\theta_1 - \theta_2) - m_2 l_1 l_2 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) + \quad (32)$$

$$+m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \sin(\theta_1 - \theta_2) - m_2 l_1 l_2 \dot{\theta}_1 \dot{\theta}_2 \sin(\theta_1 - \theta_2) + m_2 g l_2 \sin \theta_2 = 0 \quad (33)$$

Co możemy skrócić do danej formy:

$$(m_1 + m_2)l_1 \ddot{\theta}_1 + m_2 l_2 \ddot{\theta}_2 \cos(\theta_1 - \theta_2) + m_2 l_2 \dot{\theta}_2^2 \sin(\theta_1 - \theta_2) + (m_1 + m_2)g \sin \theta_1 = 0 \quad (34)$$

$$l_2 \ddot{\theta}_2 + l_1 \ddot{\theta}_1 \cos(\theta_1 - \theta_2) - l_1 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) + g \sin \theta_2 = 0 \quad (35)$$

Równania te tworzą układ nieliniowych równań różniczkowych drugiego rzędu. Możemy je uporządkować dzieląc pierwsze równanie przez  $(m_1 + m_2)l_1$ , a drugie przez  $l_2$ . Następnie przenosząc odpowiednie czynniki na daną stronę równania dostajemy następujący układ:

$$\ddot{\theta}_1 + \alpha_1(\theta_1, \theta_2) \ddot{\theta}_2 = f_1(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) \quad (36)$$

$$\ddot{\theta}_2 + \alpha_2(\theta_1, \theta_2) \ddot{\theta}_1 = f_2(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) \quad (37)$$

gdzie:

$$\alpha_1(\theta_1, \theta_2) := \frac{l_2}{l_1} \left( \frac{m_2}{m_1 + m_2} \right) \cos(\theta_1 - \theta_2) \quad (38)$$

$$\alpha_2(\theta_1, \theta_2) := \frac{l_1}{l_2} \cos(\theta_1 - \theta_2) \quad (39)$$

$$f_1(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) := -\frac{l_2}{l_1} \left( \frac{m_2}{m_1 + m_2} \right) \dot{\theta}_2^2 \sin(\theta_1 - \theta_2) - \frac{g}{l_1} \sin \theta_1 \quad (40)$$

$$f_2(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) := \frac{l_1}{l_2} \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) - \frac{g}{l_2} \sin \theta_2 \quad (41)$$

Układ ten możemy zapisać w postaci macierzowej

$$A \begin{pmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{pmatrix} = \begin{pmatrix} 1 & \alpha_1 \\ \alpha_2 & 1 \end{pmatrix} \begin{pmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} \quad (42)$$

Sprawdźmy, czy macierz  $A$  jest odwracalna

$$\det(A) = 1 - \alpha_1\alpha_2 = 1 - \left(\frac{m_2}{m_1 + m_2}\right) \cos^2(\theta_1 - \theta_2) > 0 \quad (43)$$

ponieważ  $\frac{m_2}{m_1 + m_2} < 1$  i  $\cos^2(x) \leq 1$  dla każdego  $x \in R$ .

Macierz  $A$  jest kwadratowa (2x2) oraz jej wyznacznik jest różny od zera, więc jest odwracalna.  $A^{-1}$  wynosi:

$$A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} 1 & -\alpha_1 \\ -\alpha_2 & 1 \end{pmatrix} = \frac{1}{1 - \alpha_1\alpha_2} \begin{pmatrix} 1 & -\alpha_1 \\ -\alpha_2 & 1 \end{pmatrix} \quad (44)$$

Ostatecznie dostajemy:

$$\begin{pmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{pmatrix} = A^{-1} \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \frac{1}{1 - \alpha_1\alpha_2} \begin{pmatrix} f_1 - \alpha_1 f_2 \\ -\alpha_2 f_1 + f_2 \end{pmatrix} \quad (45)$$

Wreszcie podstawiając  $\omega_1 := \dot{\theta}_1$  i  $\omega_2 := \dot{\theta}_2$ , dostajemy równanie opisujące ruch podwójnego wahadła jako układ równań różniczkowych pierwszego rzędu względem zmiennych  $\theta_1, \theta_2, \omega_1, \omega_2$ :

$$\frac{d}{dt} \begin{pmatrix} \theta_1 \\ \theta_2 \\ \omega_1 \\ \omega_2 \end{pmatrix} = \begin{pmatrix} \omega_1 \\ \omega_2 \\ g_1(\theta_1, \theta_2, \omega_1, \omega_2) \\ g_2(\theta_1, \theta_2, \omega_1, \omega_2) \end{pmatrix} \quad (46)$$

gdzie:

$$g_1 := \frac{f_1 - \alpha_1 f_2}{1 - \alpha_1\alpha_2} \quad (47)$$

$$g_2 := \frac{-\alpha_2 f_1 + f_2}{1 - \alpha_1\alpha_2} \quad (48)$$

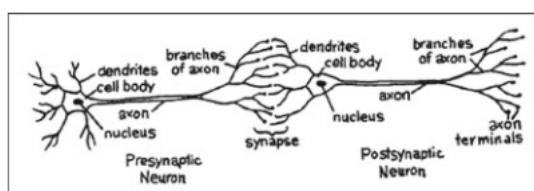
$\alpha_i = \alpha_i(\theta_1, \theta_2)$  i  $f_i = f_i(\theta_1, \theta_2, \omega_1, \omega_2)$  dla  $i = 1, 2$  zdefiniowanych w równaniach.

Powstałe równanie możemy rozwiązać korzystając z metod numerycznych. W naszych modelach użyjemy do tego celu metody Rungego-Kutty

### 0.1.5. Zadania

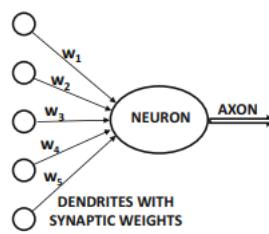
## 0.2. Sieci neuronowe

[1] Sztuczne sieci neuronowe to popularne techniki uczenia maszynowego, które naśladują sposób, w jaki uczą się organizmy biologiczne. Nasz układ nerwowy składa się z neuronów, które są ze sobą połączone aksonami i dendrytami. Miejsca, w których te połączenia się spotykają, nazywamy synapsami. Siła tych połączeń zmienia się pod wpływem bodźców zewnętrznych, co sprawia, że organizm się uczy.



Rysunek 2: Biologiczne sieci neuronowe.

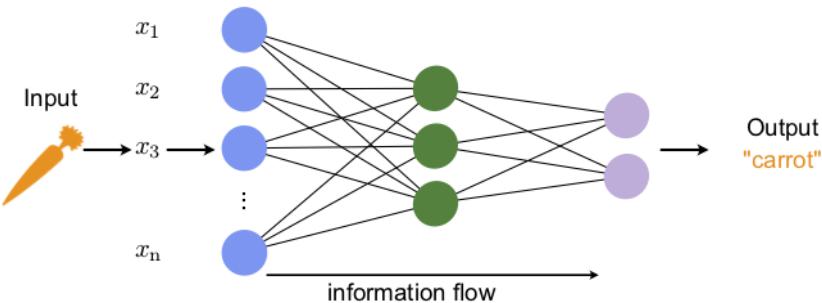
Podobny mechanizm działa w sztucznych sieciach neuronowych, gdzie mamy tzw. "neurny" – jednostki obliczeniowe. W całej pracy, gdy mówimy o „sieciach neuronowych”, mamy na myśli te sztuczne, a nie biologiczne. Neurony w sieci łączą się ze sobą za pomocą wag, które pełnią podobną rolę jak siły synaptyczne u żywych organizmów. Każde wejście do neuronu jest mnożone przez wagę, co wpływa na wynik obliczeń tej jednostki. Sieć neuronowa przetwarza dane, przekazując wartości od wejściowych neuronów do wyjściowych, używając wag jako kluczowych parametrów. Uczenie się polega na dostosowywaniu tych wag poprzez odpowiednie algorytmy.



Rysunek 3: Sztuczne sieci neuronowe.

Tak jak w przypadku organizmów potrzebne są bodźce zewnętrzne, aby się uczyć, w sztucznych sieciach tym bodźcem są dane treningowe – czyli przykłady, na których sieć ma się

nauczyć. Przykładem może być zestaw obrazów (wejście) i ich przypisanych etykieta, np. "marchewka" albo "banan" (wyjście). Sieć na podstawie obrazów próbuje przewidzieć etykiety, a dane treningowe dają jej informację zwrotną, czy zrobiła to dobrze. Jeśli sieć popełnia błędy, można to porównać do „negatywnej” informacji zwrotnej w biologii, która prowadzi do zmiany sił synaptycznych, natomiast tutaj zmieniamy wagi.



Rysunek 4: Reprezentacja etykietyzacji obrazu marchewki za pomocą sieci neuronowych.

Celem jest tak dostosować wagi, żeby sieć coraz lepiej przewidywała poprawne odpowiedzi. Wagi zmienia się w przemyślany sposób, opierając się na matematyce, żeby zmniejszyć błędy w przyszłych prognozach. Z czasem, po wielu treningach na różnych parach danych wejściowych i wyjściowych, sieć staje się coraz lepsza w przewidywaniu. Na przykład, po odpowiednim treningu na obrazach bananów, sieć nauczy się rozpoznawać banana na nowym, wcześniejszej nieznanym obrazie. Ta zdolność sieci do poprawnego przewidywania na podstawie nowych danych, na których nie była wcześniej trenowana, nazywa się uogólnianiem. Właśnie na tym polega główna siła wszystkich modeli uczenia maszynowego – potrafią wyciągać wnioski z danych, na których były trenowane, i stosować je do nowych przypadków.

### 0.2.1. Architektura sieci

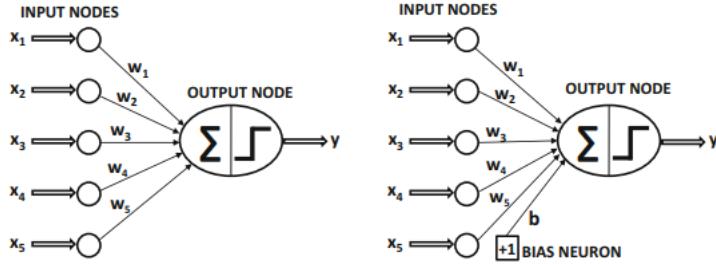
[1] Najprostszą siecią neuronową jest perceptron, składa się z jednej warstwy wejściowej i jednego węzła wyjściowego. Jego podstawowa architektura, umożliwia klasyfikację binarną. Każda instancja treningowa ma postać  $(X, y)$ , gdzie

$$X = [x_1, x_2, \dots, x_d]$$

zawiera  $d$  cech, a

$$y \in \{-1, +1\}$$

to wartość binarnej zmiennej klasowej.



Rysunek 5: Podstawowa architektura perceptronu.

Architektura perceptronu jest pokazana na Rysunku(5) , gdzie pojedyncza warstwa wejściowa przesyła cechy do węzła wyjściowego. Krawędzie z warstwy wejściowej do węzła wyjściowego zawierają wagi  $w_1 \dots w_d$ , którymi są mnożone i sumowane cechy w węźle wyjściowym. W perceptronie warstwa wejściowa składa się z  $d$  węzłów, które przesyłają cechy  $X$  za pomocą wag

$$W = [w_1, w_2, \dots, w_d]$$

do węzła wyjściowego. Obliczana jest funkcja liniowa

$$W \cdot X = \sum_{i=1}^d w_i x_i$$

a znak tej wartości określa przewidywaną wartość zmiennej  $y$ . Wynik tej predykcji jest wyrażony równaniem:

$$\hat{y} = \text{sign}(W \cdot X) = \text{sign} \left( \sum_{j=1}^d w_j x_j \right)$$

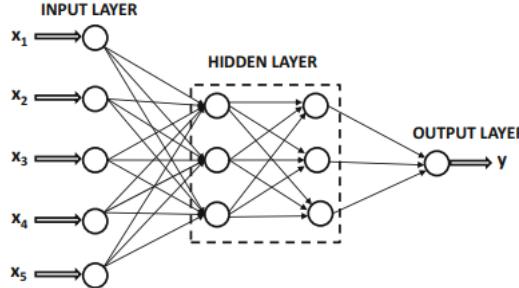
Gdzie  $\hat{y}$  jest wartością oczekiwana, a funkcja aktywacji sign mapuje wynik funkcji liniowej na  $+1$  lub  $-1$ , co jest odpowiednie do klasyfikacji binarnej. Dodatkowo często wprowadzana jest zmienna bias  $b$ , aby uwzględnić przesunięcie w danych.

$$\hat{y} = \text{sign}(W \cdot X + b) = \text{sign} \left( \sum_{j=1}^d w_j x_j + b \right)$$

Pomimo swojej prostoty, perceptron jest uważany za sieć jednowarstwową, ponieważ zawiera jedną warstwę obliczeniową (pomijając warstwę wejściową, która nie wykonuje obliczeń). W bardziej zaawansowanych architekturach perceptron stanowi fundament dla wielowarstwowych sieci neuronowych, które mogą rozwiązywać bardziej złożone problemy.

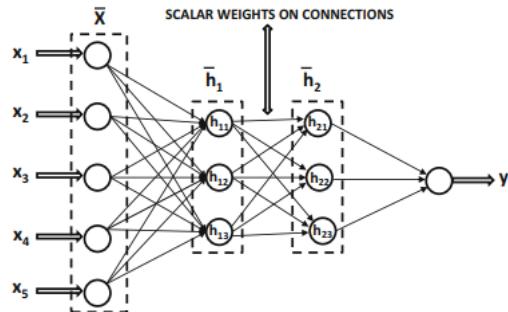
W wielowarstwowych sieciach mamy dodatkowe warstwy pomiędzy wejściem a wyjściem,

nazywane warstwami ukrytymi, ponieważ to, co tam się dzieje, nie jest widoczne dla użytkownika. Sieci, gdzie dane przepływają od wejścia do wyjścia w jednym kierunku, nazywamy sieciami jednokierunkowymi (feed-forward).



Rysunek 6: Podstawowa architektura sieci feed-forward z dwiema warstwami ukrytymi i jedną warstwą wyjściową.

Domyślana architektura sieci jednokierunkowych zakłada, że wszystkie węzły w jednej warstwie są połączone z węzłami w następnej warstwie. W związku z tym, architektura sieci neuronowej jest niemal w pełni określona, gdy zdefiniowane są liczba warstw oraz liczba/rodzaj węzłów w każdej warstwie. Każda warstwa ma określoną wymiarowość, która odpowiada liczbie neuronów w tej warstwie.



Rysunek 7: Notacja skalarna oraz podstawowa architektura sieci feed-forward z dwiema warstwami ukrytymi i jedną warstwą wyjściową.

Jeśli mamy  $k$  warstw ukrytych, a każda z tych warstw ma odpowiednio  $p_1, p_2, \dots, p_k$  neuronów, to ich wyjścia są reprezentowane przez wektory  $h_1, h_2, \dots, h_k$ , gdzie wymiar każdego wektora odpowiada liczbie neuronów w danej warstwie. Wagi połączeń między warstwą wejściową a pierwszą warstwą ukrytą znajdują się w macierzy  $W_1$  o rozmiarze  $d \times p_1$ , gdzie  $d$  liczba neuronów w warstwie wejściowej. Wagi połączeń między  $r$ -tą a  $r+1$ -szą warstwą ukrytą znajdują się w macierzy  $W_{r+1}$  o rozmiarze  $p_r \times p_{r+1}$ . Natomiast wagi połączeń mię-

czy  $k$ -tą warstwą ukrytą, a wartością wyjściową znajdują się w macierzy  $W_{k+1}$  o rozmiarze  $p_k \times o$ , gdzie  $o$  to liczba neuronów w warstwie wyjściowej. Wtedy wyjścia warstw możemy obliczyć rekurencyjnie:

Wyjście pierwszej warstwy ukrytej jest opisane jako:

$$h_1 = \Phi(W_1^T x)$$

gdzie  $W_1^T$  oznacza transponowaną macierz wag, a  $\Phi$  to funkcja aktywacji.

Dla każdej warstwy  $p \in \{1, \dots, k-1\}$  wyjście warstwy  $p+1$  jest opisane jako:

$$h_{p+1} = \Phi(W_{p+1}^T h_p), \quad \forall p \in \{1, \dots, k-1\}$$

gdzie  $W_{p+1}^T$  to transponowana macierz wag, a  $\Phi$  to funkcja aktywacji.

Wyjście warstwy wyjściowej jest opisane jako:

$$o = \Phi(W_{k+1}^T h_k)$$

gdzie  $W_{k+1}^T$  to transponowana macierz wag, a  $\Phi$  to funkcja aktywacji.

### 0.2.2. Funkcja aktywacji

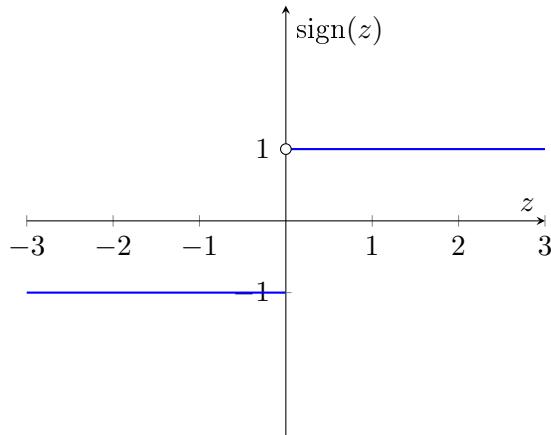
[5] "Funkcja aktywacji to funkcja nieliniowa, która przekształca lub kompresuje zważone i zsumowane wartości w węźle. Pomaga to sieci neuronowej efektywnie rozdzielić dane w celu ich klasyfikacji. Jeśli nie masz funkcji aktywacji, Twoje ukryte warstwy nie będą produktywne i nie będą działać lepiej niż regresja liniowa."

[1] Każda z następujących funkcji ma specyficzne właściwości, które sprawiają, że nadaje się do różnych zadań.

Funkcja signum zwraca wartości +1 lub -1, w zależności od znaku argumentu:

$$\text{sign}(z) = \begin{cases} +1 & \text{dla } z \geq 0, \\ -1 & \text{dla } z < 0. \end{cases}$$

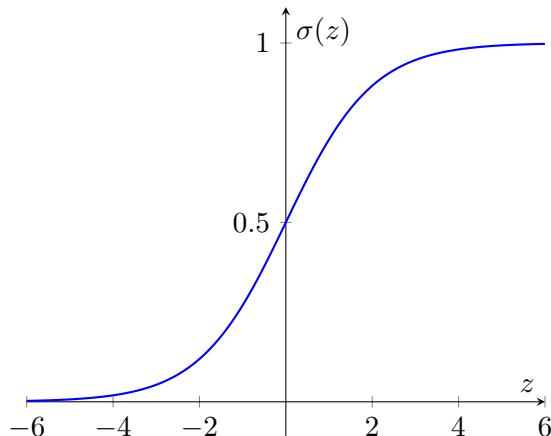
Często używana jest do klasyfikacji binarnej, ponieważ może łatwo mapować wyjście sieci neuronowej na jedną z dwóch klas (+1 lub -1). Jednak jej nieróżniczkowalność uniemożliwia efektywne trenowanie modeli za pomocą algorytmów gradientowych.



Rysunek 8: Wykres funkcji signum

Funkcja sigmoidalna jest zdefiniowana jako:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

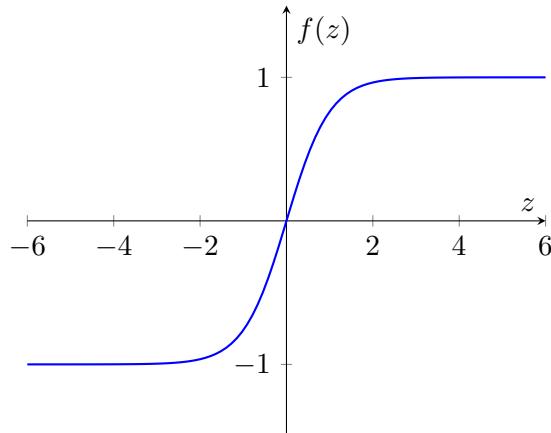


Rysunek 9: Wykres funkcji sigmoidalnej

Jej wartości są ograniczone do przedziału  $(0, 1)$ . Z tego powodu jest przydatna, gdy chcemy uzyskać wyniki, które mogą być interpretowane jako prawdopodobieństwa.

Tangens hiperboliczny (tgh) jest zdefiniowany jako:

$$\text{tgh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

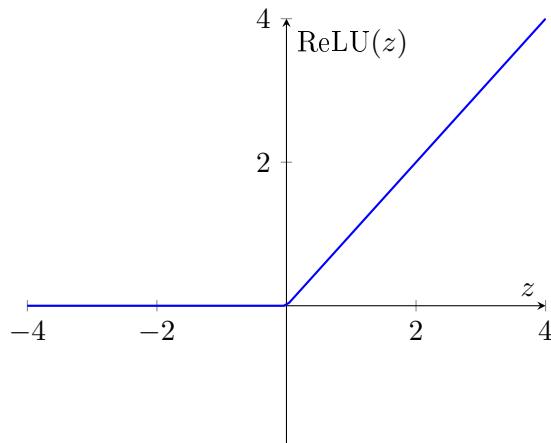


Rysunek 10: Wykres funkcji tgh

Jej wartości są ograniczone do przedziału  $[-1, 1]$ , co sprawia, że jest bardziej efektywna przy modelowaniu danych z rozkładem symetrycznym wokół zera.

Funkcja ReLU zmienia wartości ujemne w 0 i jest zdefiniowana jako:

$$\text{ReLU}(z) = \max(0, z)$$



Rysunek 11: Wykres funkcji ReLU

Jest szybsza niż funkcja sigmoidalna i tgh, łagodzi problem zanikającego gradientu i jest tania obliczeniowo.

Funkcja aktywacji softmax jest unikalna, ponieważ jest niemal zawsze stosowana w warstwie wyjściowej do przekształcania  $k$  wartości rzeczywistych na  $k$  prawdopodobieństw zdarzeń dyskretnych. Na przykład, rozważmy problem klasyfikacji  $k$ -klasowej, w którym

każdy rekord danych musi zostać przypisany do jednej z  $k$  nieuporządkowanych etykiet klasowych. W takich przypadkach można wykorzystać  $k$  wartości wyjściowych, stosując funkcję aktywacji softmax w odniesieniu do  $k$  rzeczywistych wyjść  $\mathbf{v} = [v_1, \dots, v_k]$  w węzłach danej warstwy. Ta funkcja aktywacji przekształca wartości rzeczywiste w prawdopodobieństwa, które sumują się do 1. Konkretnie, funkcja aktywacji dla  $i$ -tego wyjścia jest zdefiniowana w następujący sposób:

$$\Phi(\mathbf{v})_i = \sum_{j=1}^k \frac{\exp(v_j)}{\exp(v_i)} \quad \forall i \in \{1, \dots, k\}$$

Wiele z tych funkcji nazywamy „squashing”, ponieważ przekształcają one wyniki z szerskiego zakresu do bardziej ograniczonych wartości. Użycie nieliniowych funkcji aktywacji jest kluczowe dla zwiększenia zdolności modelowania sieci. Gdyby sieć korzystała tylko z liniowych aktywacji, nie byłaby w stanie modelować skomplikowanych wzorców lepiej niż prosta, jednowarstwowa sieć liniowa.

### 0.2.3. Funkcja kosztu

[1] Celem algorytmu optymalizacyjnego jest głównie zmniejszenie błędów w prognozach modelu. Aby to osiągnąć, wykorzystuje się tzw. funkcję kosztu, która jest miarą tego, jak bardzo przewidywania modelu odbiegają od rzeczywistych wyników. Wybór odpowiedniej funkcji kosztu zależy od rodzaju danych, które analizujemy, oraz od konkretnego zastosowania sieci neuronowej.

Przy przewidywaniu wartości liczbowych, używa się prostej straty kwadratowej dla każdej z próbek w postaci:

$$L = \sum_i^n (y_i - \hat{y}_i)^2$$

gdzie  $y$  to rzeczywista wartość, a  $\hat{y}$  to prognoza.

Istnieją jednak inne typy funkcji strat, jak np. strata zawiasowa (*hinge loss*), która dobrze działa w zadaniach klasyfikacji binarnej. Dla wartości etykiety  $y \in \{-1, +1\}$  oraz prognozy  $\hat{y}$ , funkcja straty dla pojedyńczej instancji ma postać:

$$L = \max(0, 1 - y \cdot \hat{y})$$

Dla prognoz probabilistycznych, funkcje straty zależą od tego, czy problem jest binarny, czy wieloklasowy.

W przypadku binarnych celów, takich jak w regresji logistycznej, zakłada się, że  $y$  jest

równe  $-1$  lub  $1$ , a prognoza  $\hat{y}$  może przybierać dowolne wartości. Strata dla jednej próbki jest definiowana jako:

$$L = \log(1 + \exp(-y \cdot \hat{y}))$$

W takim przypadku można także użyć funkcji aktywacji sigmoidalnej, aby uzyskać prognozy w zakresie  $(0, 1)$ .

W jaki sposób zapisać entropię krzyżową?

Ważne jest, aby pamiętać, że wybór funkcji aktywacji, funkcji straty i typu węzłów wyjściowych zależy od konkretnej aplikacji, a te elementy są ze sobą ścisłe powiązane. W rzeczywistych zastosowaniach, do problemów z wartościami dyskretnymi częściej używa się softmax z entropią krzyżową, a do wartości ciągłych – aktywacji liniowej ze strata kwadratową.

#### 0.2.4. Algorytmy optymalizacyjne

[6] Głównym celem podczas trenowania sieci neuronowej jest znalezienie takich wag i przesunięć, które minimalizują funkcję kosztu. Rozważmy problem minimalizacji ogólnej funkcji  $C(v)$ , gdzie  $v = v_1, v_2, \dots$  to zbiór zmiennych.

Naszym celem jest znalezienie miejsca, w którym funkcja  $C$  osiąga swoje minimum globalne. Możemy sobie wyobrazić tę funkcję jako dolinę, a naszą kulę, która toczy się wzduż zbocza, poszukując najniższego punktu. Wyobraźmy sobie teraz, że przesuwamy tę kulę o małe wartości  $\Delta v_1$  wzduż osi  $v_1$  i  $\Delta v_2$  wzduż osi  $v_2$ . Wówczas zmiana wartości funkcji  $C$  będzie opisana równaniem:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.$$

Naszym celem jest wybór takich wartości  $\Delta v_1$  i  $\Delta v_2$ , aby  $\Delta C$  było ujemne, co oznacza, że funkcja będzie malała, a kula toczyła się w dół doliny. Możemy to osiągnąć poprzez zastosowanie metody gradientu, która określa, jak zmieniać zmienne  $v$ , aby minimalizować  $C$ .

Definiujemy wektor zmian  $v$  jako  $\Delta v \equiv \begin{pmatrix} \Delta v_1 \\ \Delta v_2 \end{pmatrix}$ , a gradient funkcji  $C$  jako  $\nabla C \equiv \begin{pmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \end{pmatrix}$ . Teraz możemy przepisać równanie dla  $\Delta C$  jako:

$$\Delta C \approx \nabla C \cdot \Delta v.$$

To wyjaśnia, dlaczego gradient odnosi zmiany w  $v$  do zmian w funkcji  $C$ . Aby zagwarantować, że  $\Delta C$  będzie ujemne, możemy ustawić:

$$\Delta v = -\eta \nabla C,$$

gdzie  $\eta$  to współczynnik uczenia się, który kontroluje wielkość kroków. Wówczas zmiana wartości funkcji kosztu wynosi:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2.$$

Ponieważ  $\|\nabla C\|^2 \geq 0$ , mamy gwarancję, że  $\Delta C \leq 0$ , czyli funkcja  $C$  będzie malała, a nie rostała, co oznacza, że będziemy stopniowo zbliżać się do jej minimum.

Zgodnie z równaniem..., obliczamy wartość dla  $\Delta v$  i aktualizujemy pozycję  $v$  w następujący sposób:

$$v \rightarrow v' = v - \eta \nabla C.$$

Oznacza to, że za każdym razem przesuwamy  $v$  o wartość  $-\eta \nabla C$ , gdzie  $\eta$  to współczynnik uczenia się, a  $\nabla C$  to gradient funkcji kosztu  $C$ . Ten proces powtarzamy wielokrotnie, stale aktualizując wartość  $v$  w kierunku malejącego gradientu. Za każdym razem, gdy stosujemy tę regułę aktualizacji, zmniejszamy wartość funkcji kosztu  $C$ , co przybliża nas do globalnego minimum.

### 0.2.5. Dodatkowe terminy

Batch

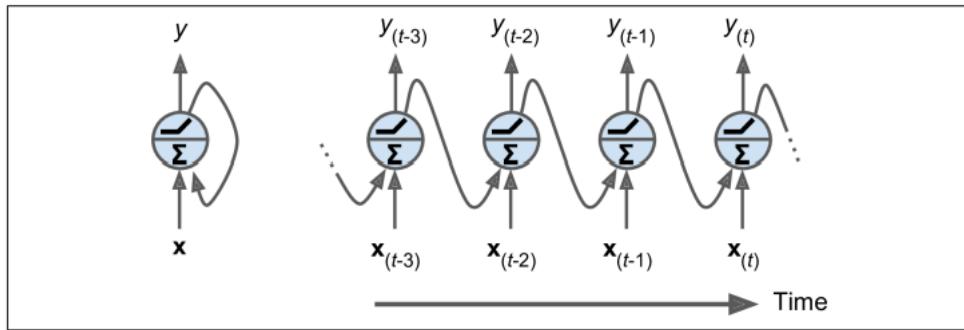
## 0.3. Rekurencyjne sieci neuronowe

### 0.3.1. Wprowadzenie

[2] W tym rozdziale przyjrzymy się rekurencyjnym sieciom neuronowym (RNN), czyli typowi sieci, które potrafią w pewnym stopniu przewidywać, co wydarzy się w przyszłości. RNN świetnie sprawdzają się w analizie danych czasowych – na przykład mogą przewidywać zmiany cen akcji i podpowiadać, kiedy warto kupić lub sprzedać. W samochodach autonomicznych są wykorzystywane do przewidywania ruchu innych pojazdów, co pomaga

w unikaniu kolizji. W przeciwieństwie do sieci, które omawialiśmy wcześniej, RNN nie są ograniczone do danych o stałej wielkości – mogą analizować sekwencje dowolnej długości, jak zdania, dokumenty czy nagrania dźwiękowe. To sprawia, że są wyjątkowo przydatne w takich zastosowaniach jak tłumaczenie maszynowe czy zamiana mowy na tekst.

### 0.3.2. Architektura

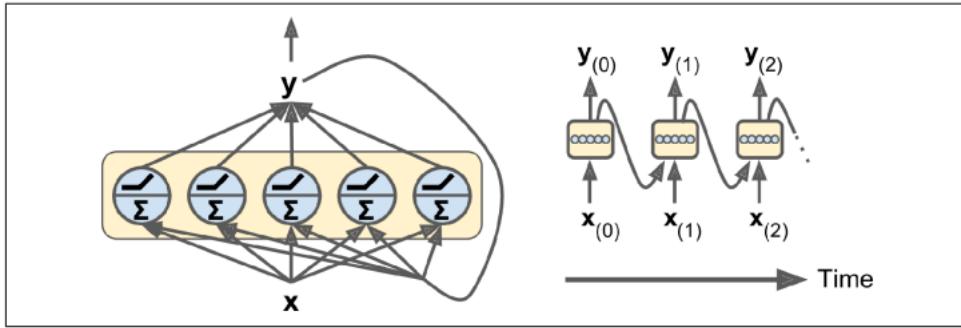


Rysunek 12: Neuron rekurencyjny (po lewej) rozwinięty w czasie (po prawej)

[2] Do tej pory skupialiśmy się na sieciach typu feedforward, gdzie sygnały przechodzą tylko w jednym kierunku – od wejścia do wyjścia. Rekurencyjne sieci neuronowe (RNN) są do nich podobne, ale oprócz połączeń naprzód mają też połączenia zwrotne. Najprostszy przykład RNN to pojedynczy neuron, który przyjmuje dane wejściowe, generuje wynik, a potem przesyła ten wynik z powrotem do siebie. Na Rysunku(12) (po lewej) pokazano, jak działa taki pojedynczy rekurencyjny neuron. W każdym kroku czasowym  $t$  neuron ten dostaje nowe dane wejściowe  $x(t)$  oraz swoje wyjście z poprzedniego kroku czasowego  $y(t-1)$ . Na początku (dla  $t = 0$ ) przyjmujemy, że wcześniejsze wyjście wynosi 0. Można przedstawić ten neuron na osi czasu, „rozwijając” go na każdy krok czasowy, co ilustruje Rysunek ... (po prawej).

Tworzenie warstwy rekurencyjnych neuronów jest dość proste. W każdym kroku czasowym  $t$  każdy neuron w warstwie otrzymuje wektor wejściowy  $x(t)$  i swoje wyjście z poprzedniego kroku,  $y(t-1)$ , co pokazano na Rysunku(13) Tutaj zarówno wejścia, jak i wyjścia to wektory (w przypadku pojedynczego neuronu mieliśmy tylko jedną wartość wyjścia, czyli skalar).

Każdy neuron RNN ma dwa zestawy wag: jeden dla wejścia  $x(t)$ , oznaczmy go  $w_x$ , oraz drugi dla wyjścia z poprzedniego kroku czasowego  $y(t-1)$ , oznaczony jako  $w_y$ . Jeśli weźmiemy pod uwagę całą warstwę, zamiast pojedynczego neuronu, wtedy możemy zebrać



Rysunek 13: Warstwa neuronów rekurencyjnych (po lewej) rozwinięta w czasie (po prawej).

wszystkie te wagi w dwie macierze:  $W_x$  i  $W_y$ . Wyjście warstwy rekurencyjnej można wtedy obliczyć zgodnie z poniższym równaniem:

$$y_t = \varphi(W_x \cdot x_t + W_y \cdot y_{t-1} + b) \quad (49)$$

gdzie  $b$  to wektor przesunięcia, a  $\varphi(\cdot)$  to funkcja aktywacji, np. ReLU.

Podobnie jak w przypadku sieci typu feedforward, można obliczyć wyjście warstwy rekurencyjnej dla całej mini-partii przypadków jednocześnie, umieszczając wszystkie wejścia z kroku czasowego  $t$  w macierzy  $X(t)$ , co prowadzi do równania:

$$Y_t = \varphi(X_t \cdot W_x + Y_{t-1} \cdot W_y + b) = \varphi([X_t | Y_{t-1}] \cdot W + b) \quad (50)$$

gdzie  $W = [W_x \ W_y]$ .

W powyższym równaniu:

- $Y(t)$  to macierz  $m \times n$ , zawierająca wyjścia warstwy w kroku czasowym  $t$  dla wszystkich przypadków z mini-partii (gdzie  $m$  to liczba przypadków, a  $n$  to liczba neuronów),
- $X(t)$  to macierz  $m \times n_w$ , zawierająca wejścia dla wszystkich przypadków (gdzie  $n_w$  to liczba cech wejściowych),
- $W_x$  to macierz  $n_w \times n$ , zawierająca wagi dla wejść z bieżącego kroku czasowego,
- $W_y$  to macierz  $n \times n$ , zawierająca wagi dla wyjść z poprzedniego kroku czasowego, -  $b$  to wektor przesunięcia o rozmiarze  $n$ .

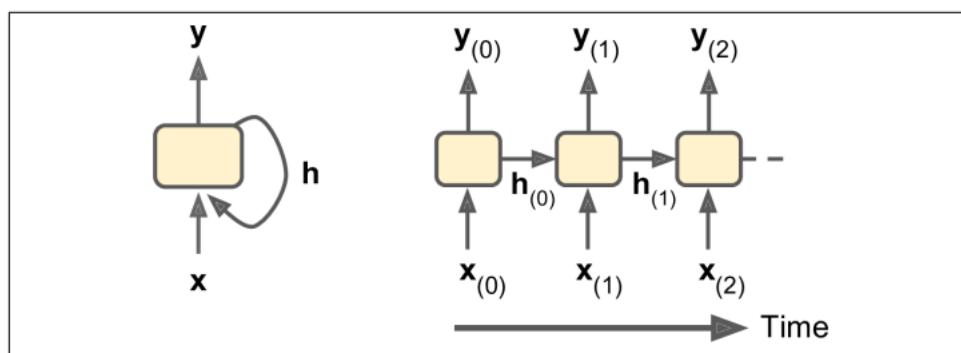
Macierze wag  $W_x$  i  $W_y$  można połączyć pionowo w jedną macierz  $W$  o rozmiarze  $(n_w + n) \times n$ , co pokazuje drugi wiersz równania (50). Notacja  $[X(t) | Y(t-1)]$  oznacza poziome połączenie macierzy  $X(t)$  i  $Y(t-1)$ .

Zauważmy, że  $Y(t)$  zależy od wartości wejściowych w poprzednich krokach czasowych, co sprawia, że wyjście w kroku  $t$  jest funkcją wszystkich wcześniejszych wejść od czasu  $t = 0$ .

### 0.3.3. Pamięć komórek

[2] Ponieważ wyjście neuronu rekurencyjnego w kroku czasowym  $t$  zależy od wszystkich wcześniejszych kroków, można powiedzieć, że posiada on pewnego rodzaju "pamięć". Część sieci, która przechowuje ten stan przez kolejne kroki, nazywamy komórką pamięci, albo po prostu komórką. Taki pojedynczy neuron rekurencyjny czy warstwa neuronów rekurencyjnych to podstawowy rodzaj komórki, która może zapamiętać jedynie krótkie wzorce (najczęściej do 10 kroków, choć może się to różnić w zależności od zadania). W dalszej części tego rozdziału omówimy bardziej zaawansowane rodzaje komórek, które radzą sobie z dłuższymi wzorcami (nawet 10 razy dłuższymi, choć i to zależy od konkretnego zadania).

Stan komórki w kroku czasowym  $t$ , oznaczany jako  $h(t)$  (gdzie „h” od „hidden” – ukryty), zależy od bieżących danych wejściowych oraz stanu z poprzedniego kroku:  $h(t) = f(h(t - 1), x(t))$ . Podobnie, wyjście komórki w kroku czasowym  $t$ , oznaczane jako  $y(t)$ , także opiera się na wcześniejszym stanie i bieżących danych wejściowych. W przypadku podstawowych komórek, które dotychczas omawialiśmy, wyjście jest po prostu równe stanowi komórki. Jednak w bardziej zaawansowanych typach komórek to wyjście może działać inaczej i nie zawsze jest dokładnie tym samym co stan komórki.



Rysunek 14: Stan ukryty komórki i jej wyjście mogą być różne.

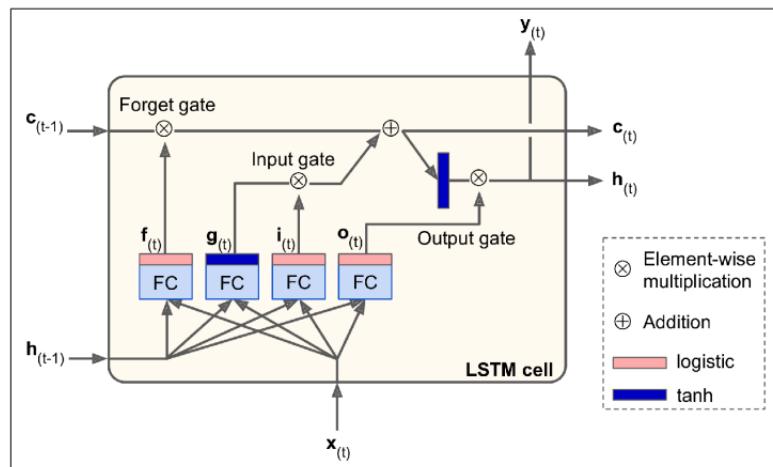
### 0.3.4. LSTM

[2] Podczas przechodzenia przez rekurencyjną sieć neuronową (RNN) dane przechodzą przez różne transformacje, co sprawia, że z każdym krokiem tracimy część informacji. Po

jakimś czasie stan RNN nie pamięta prawie nic z pierwszych danych wejściowych, co może być sporym problemem. Wyobraź sobie, że Dory, rybka, stara się przetłumaczyć długie zdanie; gdy w końcu kończy czytanie, nie ma pojęcia, jak to zdanie brzmiało na początku. Aby rozwiązać ten problem, wymyślono różne typy komórek z pamięcią długoterminową. Spisały się one na tyle dobrze, że podstawowe komórki są teraz rzadko stosowane. Zobaczmy więc najpopularniejszy z tych typów komórek z pamięcią długoterminową: komórkę LSTM.

Jeśli potraktujesz komórkę LSTM jak czarną skrzynkę, to możesz ją używać podobnie jak zwykłą komórkę, ale będzie działać znacznie lepiej. Szkolenie przebiega szybciej, a LSTM potrafi uchwycić długoterminowe zależności w danych.

Jak działa komórka LSTM? Jej budowę możesz zobaczyć na Rysunku(15)



Rysunek 15: Komórka LSTM

Na pierwszy rzut oka, komórka LSTM wygląda jak zwykła komórka, z tym, że jej stan jest podzielony na dwa wektory:  $h(t)$  i  $c(t)$  (gdzie „c” oznacza „komórka” - „cell”). Możesz myśleć o  $h(t)$  jako o stanie krótkoterminowym, a  $c(t)$  jako o stanie długoterminowym. Teraz otwórzmy tę czarną skrzynkę! Kluczową ideą jest to, że sieć może nauczyć się, co przechowywać w stanie długoterminowym, co wyrzucić i co z niego odczytać. Kiedy stan długoterminowy  $c(t-1)$  przemieszcza się przez sieć, najpierw trafia do bramki zapomnienia, która usuwa niektóre wspomnienia, a potem dodaje nowe wspomnienia przez operację dodawania (gdzie nowe wspomnienia są wybierane przez bramkę wejściową). Wynik  $c(t)$  jest przesyłany dalej bez dodatkowych zmian. Tak więc w każdym kroku czasowym niektóre wspomnienia są usuwane, a inne dodawane. Dodatkowo, po operacji dodawania,

stan długoterminowy jest kopiowany, przetwarzany przez funkcję tanh, a potem wynik jest filtrowany przez bramkę wyjściową, co daje nam stan krótkoterminowy  $h(t)$  (który jest równy wyjściu komórki w tym kroku czasowym,  $y(t)$ ).

Teraz zobaczymy, skąd biorą się nowe wspomnienia i jak działają bramki. Najpierw bieżący wektor wejściowy  $x(t)$  i poprzedni stan krótkoterminowy  $h(t-1)$  są przesyłane do czterech różnych warstw, z których każda ma swoją rolę:

Główna warstwa generuje  $g(t)$ . Odpowiada za analizę bieżących wejść  $x(t)$  i poprzedniego stanu  $h(t-1)$ . W podstawowej komórce ta warstwa była jedyną, a jej wyjście szło bezpośrednio do  $y(t)$  i  $h(t)$ . W komórce LSTM wyjście tej warstwy nie jest przesyłane od razu, ale kluczowe informacje są zapisywane w stanie długoterminowym, a reszta jest usuwana. Pozostałe trzy warstwy to kontrolery bramek. Działają one na zasadzie funkcji aktywacji logistycznej, więc ich wyjścia wahają się od 0 do 1. Jak widać, ich wyjścia są używane do mnożenia elementów, więc jeśli wyjście wynosi 0, bramka jest zamknięta, a jeśli 1, to otwarta. Oto ich konkretne role:

Bramka zapomnienia (sterowana przez  $f(t)$ ) decyduje, które części stanu długoterminowego mają być usunięte. Bramka wejściowa (sterowana przez  $i(t)$ ) kontroluje, które części  $g(t)$  mają być dodane do stanu długoterminowego.

Bramka wyjściowa (sterowana przez  $o(t)$ ) decyduje, które części stanu długoterminowego mają być odczytane i przesyłane jako wyjście w tym kroku czasowym, zarówno do  $h(t)$ , jak i do  $y(t)$ .

Podsumowując, komórka LSTM potrafi nauczyć się rozpoznawać ważne dane wejściowe (to zadanie bramki wejściowej), przechowywać je w stanie długoterminowym, utrzymywać je tak długo, jak to konieczne (to rola bramki zapomnienia) i wydobywać je w razie potrzeby. To sprawia, że te komórki są niezwykle skuteczne w uchwyceniu długoterminowych wzorców w szeregach czasowych, długich tekstach, nagraniach audio i innych danych.

Poniższe równania podsumowują, jak obliczyć stan długoterminowy komórki, jej stan krótkoterminowy i wyjście w każdym kroku czasowym dla pojedynczego przypadku (równania dla całego mini-batcha są bardzo podobne).

$$i_t = \sigma(W_{xi} \cdot x_t + W_{hi} \cdot h_{t-1} + b_i) \quad (51)$$

$$f_t = \sigma(W_{xf} \cdot x_t + W_{hf} \cdot h_{t-1} + b_f) \quad (52)$$

$$o_t = \sigma(W_{xo} \cdot x_t + W_{ho} \cdot h_{t-1} + b_o) \quad (53)$$

$$g_t = \tanh(W_{xg} \cdot x_t + W_{hg} \cdot h_{t-1} + b_g) \quad (54)$$

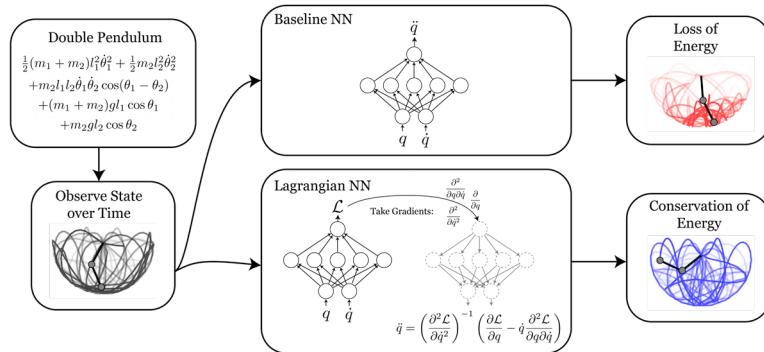
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (55)$$

$$h_t = o_t \odot \text{tgh}(c_t) \quad (56)$$

W tym równaniu:

- $W_{xi}, W_{xf}, W_{xo}, W_{xg}$  to macierze wag każdej z czterech warstw dla ich połączenia z wektorem wejściowym  $x(t)$ .
- $W_{hi}, W_{hf}, W_{ho}, W_{hg}$  to macierze wag każdej z czterech warstw dla ich połączenia z poprzednim stanem krótkoterminowym  $h(t-1)$ .
- $b_i, b_f, b_o, b_g$  to składniki biasu dla każdej z czterech warstw. Należy zauważyć, że TensorFlow inicjalizuje  $b_f$  jako wektor pełen jedynek zamiast zer. Zapobiega to zapomnieniu wszystkiego na początku treningu.

#### 0.4. Lagranżowskie sieci neuronowe



Lagranżowskie sieci neuronowe (LNN) to specyficzny rodzaj sieci neuronowych, które wykorzystują zasady mechaniki lagrangowskiej do modelowania i przewidywania ruchów fizycznych systemów. Dzięki temu mogą być bardziej dokładne i zrozumiałe w kontekście fizycznym. Jak dobrze wiemy z pierwszego rozdziału mechanika lagranżowska to podejście do opisywania ruchu obiektów. Główna idea polega na użyciu funkcji Lagrange'a  $L$ , która jest różnicą między energią kinetyczną  $T$ , a energią potencjalną  $V$ :  $L = T - V$ . Za pomocą tej funkcji możemy następująco wyprowadzić równania ruchu dla systemu, używając równań Eulera-Lagrange'a:

$$\frac{d}{dt} \left( \frac{\partial L}{\partial \dot{q}_i} \right) - \frac{\partial L}{\partial q_i} = 0 \quad (57)$$

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{q}_i} = \frac{\partial L}{\partial q_i} \quad (58)$$

Zamieniamy na notację wektorową:

$$\frac{d}{dt} \nabla_{\dot{q}} L = \nabla_q L \quad (59)$$

Następnie liczymy pochodną (wy tłumaczyć)

$$(\nabla_{\dot{q}} \nabla_{\dot{q}}^\top L) \ddot{q} + (\nabla_q \nabla_{\dot{q}}^\top L) \dot{q} = \nabla_q L \quad (60)$$

Na końcu porządkujemy równanie, by obliczyć  $\ddot{q}$ :

$$\ddot{q} = (\nabla_{\dot{q}} \nabla_{\dot{q}}^\top L)^{-1} [\nabla_q L - (\nabla_q \nabla_{\dot{q}}^\top L) \dot{q}] \quad (61)$$

Tak więc zamiast od razu przewidywać przyspieszenie lub współrzędne układu, LNN modeluje za pomocą sieci neuronowej funkcje Lagrange'a. Sieć przyjmuje jako wejścia współrzędne uogólnione i prędkości a jako wyjście zwraca wartość funkcji Lagrange'a. Na końcu model oblicza z niej potrzebne nam przyspieszenie. Dzięki temu sieć uczy się dynamicznego modelu systemu na podstawie danych fizycznych. Podczas treningu sieć neuronowa optymalizuje poprzez funkcję kosztu swoje parametry, aby minimalizować różnicę między przewidywanymi a rzeczywistymi wartościami.

## 0.5. Badania

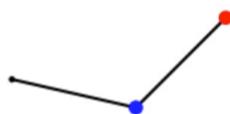
Model bez LNN, Model LNN z danymi numerycznymi, model LNN z danymi rzeczywistymi

### 0.5.1. Uzyskanie danych za pomocą metod numerycznych

W pierwszej kolejności ustalono kąty oraz prędkości początkowe.

```
x0 = np.array([3*np.pi/7, 3*np.pi/4, 0, 0], dtype=np.float32)
```

W podanym przykładzie pierwszy element wahadła tworzy kąt  $\frac{3}{7}\pi$  z linią pionową, natomiast drugi element kąt  $\frac{3}{4}\pi$ . Obie części wahadła charakteryzują się zerową prędkością początkową.



Następnie, przy użyciu funkcji odeint, która bazuje na metodzie Rungego-Kutty w celu rozwiązyania układów równań różniczkowych zwyczajnych, obliczono równanie (7). W efekcie uzyskano wartości kątów oraz prędkości kątowych dla przyjętego kroku czasowego t.

```
t = np.linspace(0, 100, num=1501)
x_train = jax.device_get(solve_analytical(x0, t))
```

Aby móc zastosować Recurrent Neural Network (RNN), konieczne jest przekształcenie danych z układu współrzędnych kątowych na współrzędne kartezjańskie. Taka transformacja umożliwia modelowi skuteczniejsze przetwarzanie danych, które w tym przypadku są związane z ruchem wahadła. Przekształcenie współrzędnych kątowych na kartezjańskie odbywa się przy użyciu zależności trygonometrycznych.

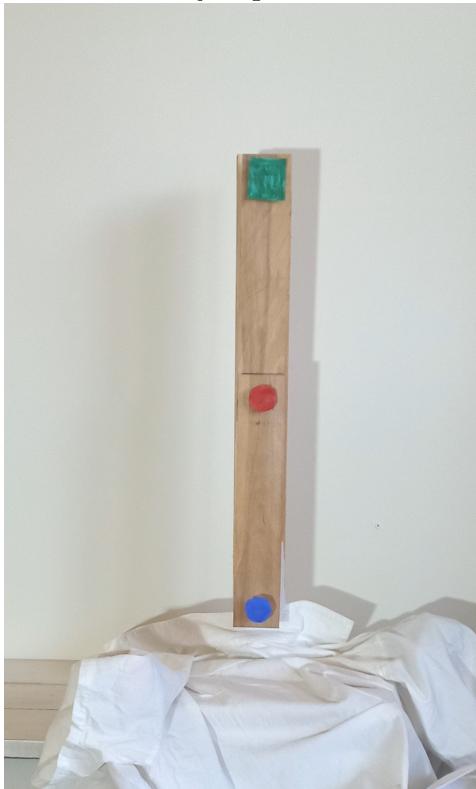
```
def to_cartesian(theta1, w1, theta2, w2, L1, L2):
    """ Transforms theta and omega to cartesian coordinates
    and velocities x1, y1, x2, y2, vx1, vy1, vx2, vy2
    """
    x1 = L1 * np.sin(theta1)
    y1 = -L1 * np.cos(theta1)
    x2 = x1 + L2 * np.sin(theta2)
    y2 = y1 - L2 * np.cos(theta2)
    vx1 = L1*np.cos(theta1)*w1
    vy1 = L1*np.sin(theta1)*w1
    vx2 = vx1 + L2*np.cos(theta2)*w2
    vy2 = vy1 + L2*np.sin(theta2)*w2
    return x1, y1, x2, y2, vx1, vy1, vx2, vy2

theta1, theta2, w1, w2 = x_train[:, 0], x_train[:, 1], x_train[:, 2], x_train[:, 3]

x1_new1, y1_new1, x2_new1, y2_new1 = to_cartesian(theta1, w1, theta2, w2, L1, L2)[:4]
```

### 0.5.2. Uzyskanie danych za pomocą estymacji pozycji rzeczywistego wahadła podwójnego

W celu realizacji tego zadania został stworzony fizyczny model podwójnego wahadła.



Na początkowym etapie z arkusza sklejki wycięto dwie części o odpowiednich wymiarach.

Następnie, przy użyciu ..., wycięto otwory o średnicy ... w celu zamontowania łożysk.



Wszystkie elementy modelu zostały dokładnie zważone i zmierzone, co pozwoliło na precyzyjne określenie parametrów potrzebnych do późniejszych obliczeń i symulacji.



W drugim ramieniu wahadła zamontowano dodatkowe śrubki w celu wyrównania masy obu ramion, co zapewnia ich odpowiedni ruch.

Zamierzamy również stworzyć program do wykrywania określonych kolorów na nagranym materiale wideo, który pozwoli na analizę ruchu wahadła. W tym celu końce ramion wahadła zostały pomalowane na kolory: zielony, czerwony oraz niebieski. Dzięki temu, po odczytaniu pozycji tych kolorów na obrazie, możliwe będzie uzyskanie współrzędnych ramion wahadła. Następnie, zarejestrowano filmy przedstawiające ruch wahadła z różnymi punktami początkowymi. Program, który analizuje te nagrania, działa w następujący sposób:

W słowniku *color\_ranges* zdefiniowano zakresy kolorów w przestrzeni HSV (Hue, Saturation, Value) dla kolorów czerwonego, zielonego oraz niebieskiego.

```
color_ranges = {
    'red': [(0, 120, 70), (10, 255, 255)],
    'green': [(36, 100, 100), (86, 255, 255)],
    'blue': [(94, 80, 2), (126, 255, 255)]
}
```

Program najpierw konwertuje każdą klatkę wideo z przestrzeni kolorów BGR do przestrzeni HSV, co umożliwia bardziej efektywne wykrywanie kolorów, ponieważ przestrzeń HSV lepiej oddaje różnice w barwach i nasyceniu niż BGR.

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Następnie tworzy maski dla zdefiniowanych wcześniej kolorów (czerwonego, zielonego i niebieskiego). Czyli piksele odpowiadające danemu kolorowi mają wartość 1, podczas gdy pozostałe piksele przyjmują wartość 0.

```
mask = cv2.inRange(hsv, lower_bound, upper_bound)
```

W kolejnym kroku program znajduje kontury obszarów, które są zgodne z maskami. Dla każdego wykrytego konturu o powierzchni większej niż 500 pikseli program wylicza współrzędne środka prostokąta otaczającego kontur. W końcowym etapie zwraca listę współrzędnych środków wszystkich wykrytych konturów, co pozwala na określenie pozycji końców ramion wahadła w każdej klatce filmu.

```
contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
centers = []

for contour in contours:
    x, y, w, h = cv2.boundingRect(contour)

    center_x = x + w // 2
    center_y = y + h // 2
    centers.append((center_x, center_y))

return centers
```

Program oblicza medianę współrzędnych środków wszystkich wykrytych obszarów dla każdego koloru (czerwonego, zielonego i niebieskiego), co pozwala na bardziej stabilne i precyzyjne wyznaczenie pozycji końców ramion wahadła, eliminując wpływ potencjalnych szumów i błędnych detekcji. Wyliczone mediany są następnie dodawane do odpowiednich list dla wszystkich kolorów. Dzięki temu uzyskujemy sekwencję współrzędnych, które odzwierciedlają ruch ramion wahadła w czasie.

```
for color_name, centers in all_coordinates.items():
    if centers:
        medians[color_name] = (
            int(np.median([c[0] for c in centers])),
            int(np.median([c[1] for c in centers])))
    else:
        medians[color_name] = (None, None)

Red center: (221, 493), Green center: (343, 400), Blue center: (106, 373)
Red center: (220, 494), Green center: (341, 402), Blue center: (96, 382)
Red center: (224, 497), Green center: (341, 402), Blue center: (87, 400)
Red center: (229, 502), Green center: (341, 402), Blue center: (75, 427)
Red center: (235, 508), Green center: (338, 403), Blue center: (78, 463)
Red center: (243, 517), Green center: (340, 406), Blue center: (82, 507)
Red center: (252, 521), Green center: (342, 402), Blue center: (94, 556)
Red center: (259, 527), Green center: (340, 406), Blue center: (None, None)
Red center: (265, 536), Green center: (338, 401), Blue center: (None, None)
```

Niestety, ze względu na szybki ruch wahadła, w niektórych miejscach program nie zdołał wykryć kolorów, co powoduje brakujące dane.

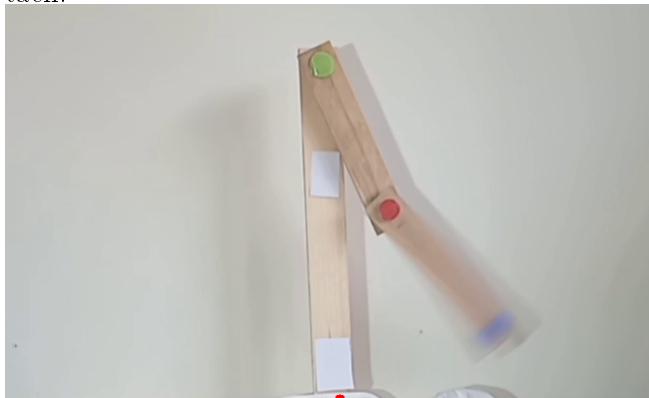
```
Red center: (259, 527), Green center: (340, 406), Blue center: (None, None)
```

Problem można rozwiązać ręcznie, jednak najpierw konieczne jest zidentyfikowanie filmu, w którym występuje najmniej niewykrytych klatek. Na pierwszy rzut oka wydaje się, że odpowiednim wyborem jest plik video3.mp4.

```
Liczba punktów nullowych niebieskiego koloru: 4
Liczba punktów nullowych czerwonego koloru: 39
```

Jednak po dokładniejszej analizie wyników programu okazuje się, że w wielu klatkach zamiast niebieskiego punktu program błędnie wykrywa białą narzutę jako niebieski kolor. To

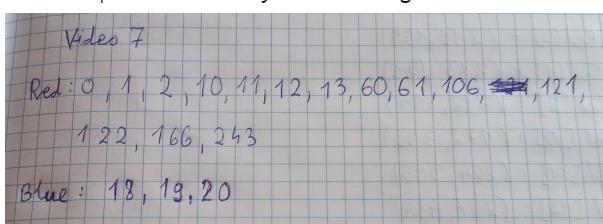
prowadzi do fałszywych detekcji i zniekształca dane o pozycjach wahadła w tych momencach.



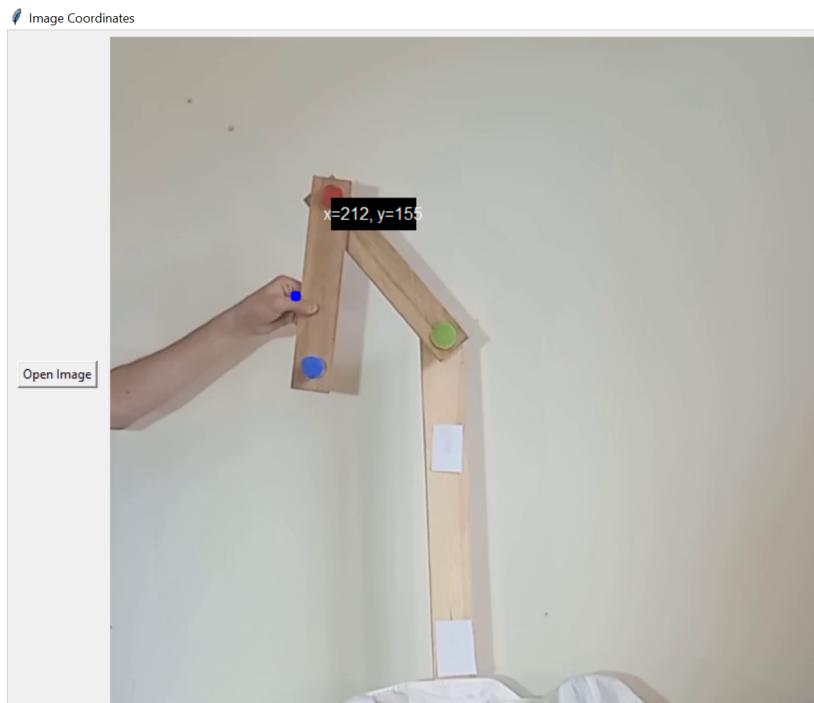
Po przeanalizowaniu kolejnych materiałów wideo, okazało się, że jedynie w 3 klatkach wartości kolorów zostały wykryte błędnie, a w 14 klatkach kolory nie zostały wykryte w ogóle. W związku z tym, konieczne jest ręczne uzupełnienie lub poprawienie danych w łącznie 17 klatkach. Zapisujemy numery tych klatek, aby móc je sprawdzić ręcznie.

```
#video7
null_points_count_blue7 = sum(1 for point in blue_centers7 if point == (None, None))
null_points_count_red7 = sum(1 for point in red_centers7 if point == (None, None))
print(len(blue_centers7))
print(f'Liczba punktów nullowych niebieskiego koloru: {null_points_count_blue7}')
print(f'Liczba punktów nullowych czerwonego koloru: {null_points_count_red7}')
```

1480  
Liczba punktów nullowych niebieskiego koloru: 3  
Liczba punktów nullowych czerwonego koloru: 11



W tym celu stworzymy w Tkinter aplikację, która umożliwia wyświetlanie współrzędnych klatek wideo. Dzięki niej ręcznie uzupełniamy brakujące dane lub korygujemy nieprawidłowo wykryte wartości.



Red [0] = (212, 155)	Red [106] = (401, 422)
Red [1] = (213, 153)	Red [121] = (324, 444)
Red [2] = (212, 155)	Red [122] = (275, 436)
Red [10] = (172, 350)	Red [166] = (286, 441)
Red [11] = (204, 395)	Red [243] = (328, 445)
Red [12] = (257, 434)	
Red [13] = (316, 445)	Blue [18] = (461, 468)
Red [60] = (248, 427)	Blue [19] = (521, 449)
Red [61] = (197, 391)	Blue [20] = (572, 423)

Po wprowadzeniu poprawek, dla pewności sprawdzamy ponownie, czy nie ma jeszcze brakujących lub błędnych wartości, aby zapewnić kompletność i poprawnosc danych.

```
null_points_count_red7 = np.sum(np.any(np.isnan(red_centers7), axis=1))
print(f'Liczba punktów nullowych czerwonego koloru: {null_points_count_red7}')
```

```
Liczba punktów nullowych czerwonego koloru: 0
null_points_count_blue7 = np.sum(np.any(np.isnan(blue_centers7), axis=1))
print(f'Liczba punktów nullowych niebieskiego koloru: {null_points_count_blue7}')
```

Liczba punktów nullowych niebieskiego koloru: 0

Zielony punkt został wybrany jako punkt zerowy układu odniesienia, dlatego aby precyzyjnie określić jego pozycję, obliczamy medianę wszystkich współrzędnych zielonego punktu z każdej klatki. Medianą ta będzie reprezentować stałą pozycję punktu zerowego.

```
green_center7.append(np.nanmedian(green_centers7[:,0]))
green_center7.append(np.nanmedian(green_centers7[:,1]))
green_center7
```

[323.0, 285.0]

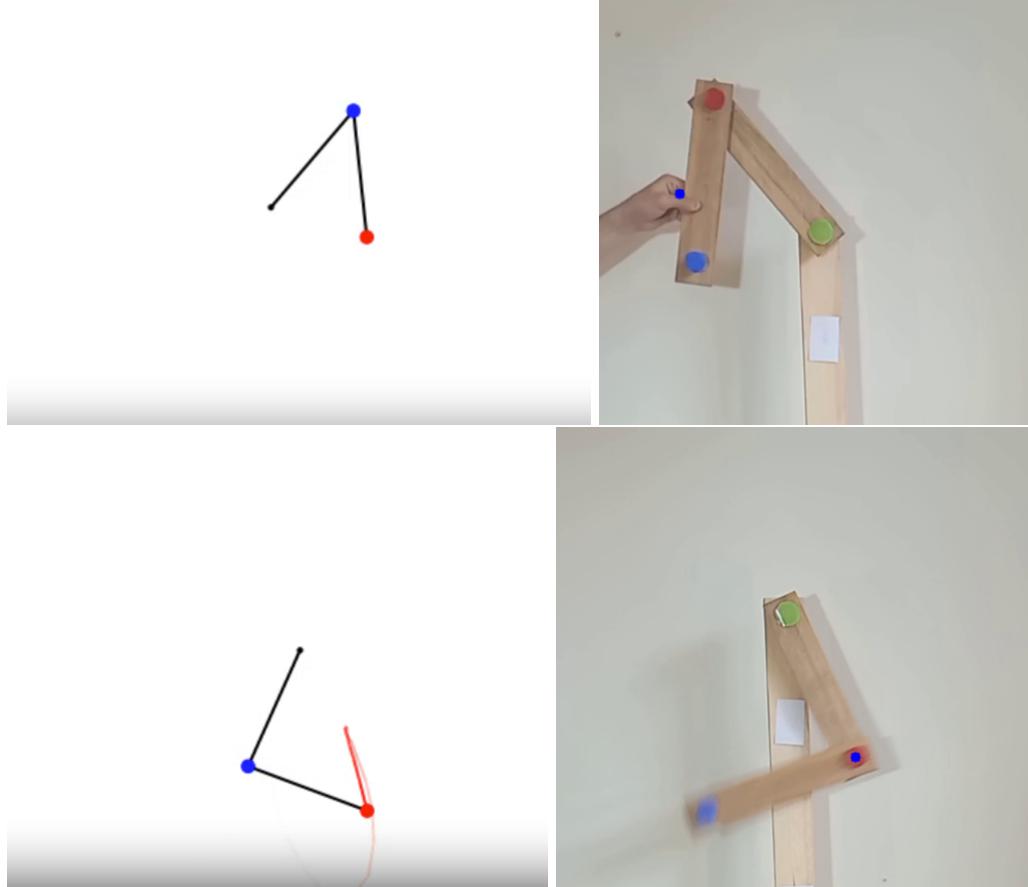
Po ustaleniu punktu odniesienia możemy przystąpić do normalizacji danych. Punkt odpowiadający pierwszemu ramieniu wahadła jest wyznaczony jako znormalizowana różnica między współrzędnymi wykrytego czerwonego punktu a współrzędnymi zielonego punktu.

```
red2=preprocessing.normalize(green_center7-red_centers7)
red2
array([[ 0.64934491,  0.76049404],
       [ 0.6401844 ,  0.76822128],
       [ 0.64646424,  0.76294429],
       ...,
       [-0.18768705, -0.98222888],
       [-0.17378533, -0.98478356],
       [-0.1505811 , -0.98859766]])
```

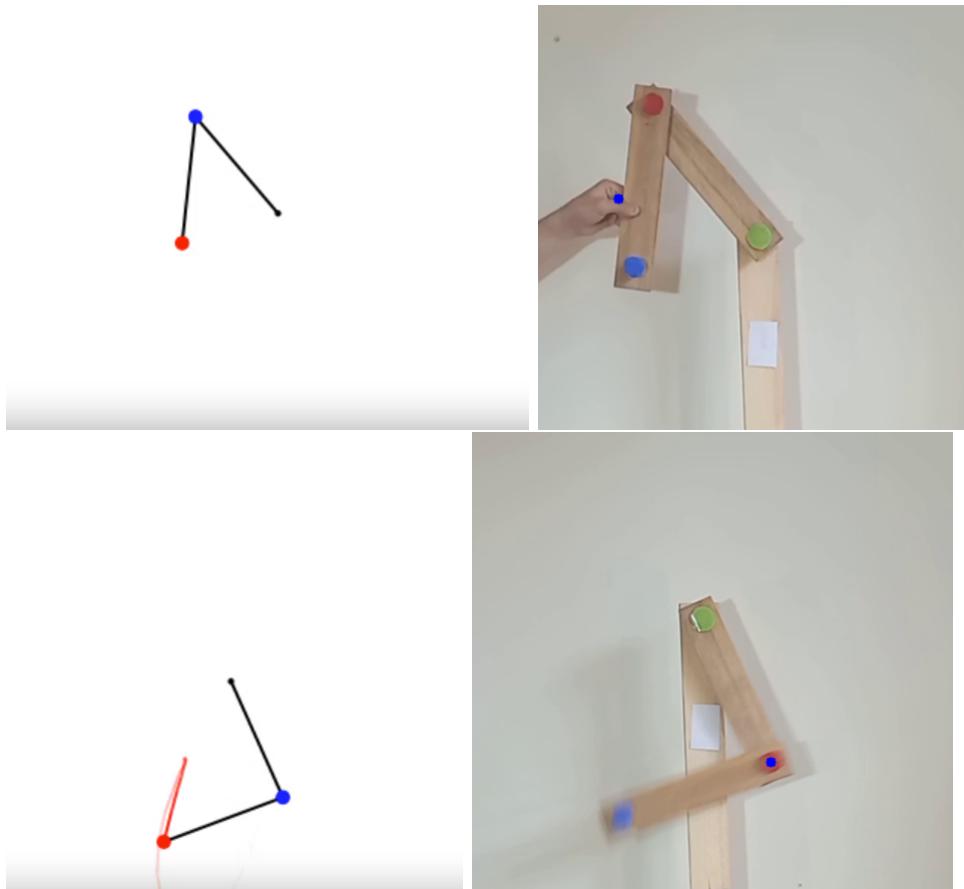
Dla drugiego ramienia wahadła punkt ten obliczamy, dodając do współrzędnych pierwszego ramienia znormalizowaną różnicę pomiędzy współrzędnymi niebieskiego punktu a współrzędnymi czerwonego punktu. Dzięki temu otrzymujemy znormalizowane współrzędne obu ramion wahadła względem punktu zerowego.

```
blue2=red2+preprocessing.normalize(red_centers7-blue_centers7)
blue2
array([[ 0.75435122, -0.23397751],
       [ 0.7559649 , -0.22505354],
       [ 0.75622646, -0.23101359],
       ...,
       [-0.58388212, -1.90039524],
       [-0.58106861, -1.89808544],
       [-0.56299227, -1.89959548]])
```

Na ostatnim etapie sprawdzono, czy utworzony model na podstawie zgromadzonych danych działa poprawnie. Okazało się, że model jest prawidłowy, jednak przedstawiony w lustrzanym odbiciu.



Aby skorygować ten problem, wystarczyło dokonać odbicia lustrzanego poprzez zmianę znaku współrzędnej  $x$  dla wszystkich punktów. Po tej korekcie model działał poprawnie, a uzyskane dane wiernie odzwierciedlały rzeczywisty ruch podwójnego wahadła, zgodnie z estymacją jego pozycji.



#### 0.5.3. Model przy użyciu RNN oparty na danych uzyskanych numerycznie, nauczony na współrzędnych

Po zebraniu i przekształceniu danych możemy przystąpić do tworzenia modelu. Korzystając z Recurrent Neural Network (RNN), musimy ustalić liczbę klatek, które sieć będzie analizowała do przewidzenia kolejnej klatki. W naszym przypadku timeframe wynosi 20, co oznacza, że sieć będzie analizowała 20 poprzednich układów współrzędnych, aby przewidzieć kolejne.

Dla tego celu przekształcamy tablicę z pierwotnymi danymi, która zawiera 1500 tablic (po 4 zmienne w każdej), w nową tablicę o rozmiarze 1480 tablic, gdzie każda ma wymiar 20x4. To oznacza, że na każde 20 zestawów współrzędnych sieć będzie próbowała przewidzieć kolejny 21. zestaw.

```

for i in range(len(x1_new) - timeframe):
    # Stack the next 'timeframe' positions of both masses to serve as the input sequence
    X.append(np.vstack((x1_new[i:i+timeframe], y1_new[i:i+timeframe], x2_new[i:i+timeframe], y2_new[i:i+timeframe])).T)

    # Record the positions of both masses 20 steps ahead from the current timestep to set as the prediction target
    Y.append([x1_new[i+timeframe], y1_new[i+timeframe], x2_new[i+timeframe], y2_new[i+timeframe]])

```

Następnie przystępujemy do tworzenia modelu. Wykorzystamy do tego LSTM (Long Short-Term Memory), który jest wariantem RNN, dobrze radzącym sobie z długoterminowymi zależnościami, oraz głęboką sieć neuronową, aby zwiększyć złożoność modelu.

```

model = Sequential([
    LSTM(64, input_shape=(timeframe, 4), return_sequences=True),
    LSTM(32), # Second LSTM Layer with 32 units
    Dense(4) # Output Layer for future x1, y1, x2, y2 positions
])

```

Działanie sieci można opisać, że sieć stara się z danych zawartych w pierwszych 20 układach przewidzieć, jak wygląda układ współrzędnych w 21. klatce. Dla przykładu, X[0] reprezentuje pierwsze 20 układów współrzędnych, natomiast Y[0] to 21. układ współrzędnych. W ten sposób sieć uczy się modelować dynamiczne zależności w ruchu wahadła i przewidywać jego przyszłe położenia na podstawie wcześniejszych stanów.

X[0]	X[1]
<code>array([[ 0.9749279, -0.22252098,  1.6820347,  0.4845858 ],        [ 0.970309 , -0.24186869,  1.6809305,  0.46170586],        [ 0.95439947, -0.2985325 ,  1.6771235 ,  0.39260423],        [ 0.92161924, -0.38809535 ,  1.6692172 ,  0.2760562 ],        [ 0.8646444 , -0.5023843 ,  1.6549351 ,  0.11034787],        [ 0.77764106, -0.6287085 ,  1.6305242 , -0.10660666],        [ 0.6611618 , -0.75024337,  1.5889895 , -0.37723434],        [ 0.5276897 , -0.84943724,  1.5168946 , -0.7028985 ],        [ 0.404249 , -0.91464895,  1.3907369 , -1.0784829 ],        [ 0.32850385, -0.94450265 ,  1.1738546 , -1.4787143 ],        [ 0.33271384, -0.94302785,  0.81444937 , -1.8193495 ],        [ 0.36055085, -0.93273956,  0.31551835, -1.931725 ],        [ 0.29231253, -0.95632285, -0.18669364, -1.8341343 ],        [ 0.11806197, -0.9930062 , -0.62938297, -1.65733 ],        [-0.13207965, -0.99123913, -1.0116328 , -1.4670398 ],        [-0.4144933 , -0.91005236, -1.3395425 , -1.2898996 ],        [-0.68307096, -0.730352 , -1.600534 , -1.600534 , -1.281732 ],        [-0.8862997 , -0.4631121 , -1.7533457 , -0.96134007],        [-0.9830793 , -0.1831804 , -1.7907777 , -0.77277637],        [-0.99896413,  0.04550415, -1.7722844 , -0.58851147]],       dtype=float32)</code>	<code>array([[ 0.970309 , -0.24186869,  1.6809305,  0.46170586],        [ 0.95439947, -0.2985325 ,  1.6771235 ,  0.39260423],        [ 0.92161924, -0.38809535 ,  1.6692172 ,  0.2760562 ],        [ 0.8646444 , -0.5023843 ,  1.6549351 ,  0.11034787],        [ 0.77764106, -0.6287085 ,  1.6305242 , -0.10660666],        [ 0.6611618 , -0.75024337,  1.5889895 , -0.37723434],        [ 0.5276897 , -0.84943724,  1.5168946 , -0.7028985 ],        [ 0.404249 , -0.91464895,  1.3907369 , -1.0784829 ],        [ 0.32850385, -0.94450265 ,  1.1738546 , -1.4787143 ],        [ 0.33271384, -0.94302785,  0.81444937 , -1.8193495 ],        [ 0.36055085, -0.93273956,  0.31551835, -1.931725 ],        [ 0.29231253, -0.95632285, -0.18669364, -1.8341343 ],        [ 0.11806197, -0.9930062 , -0.62938297, -1.65733 ],        [-0.13207965, -0.99123913, -1.0116328 , -1.4670398 ],        [-0.4144933 , -0.91005236, -1.3395425 , -1.2898996 ],        [-0.68307096, -0.730352 , -1.600534 , -1.600534 , -1.281732 ],        [-0.8862997 , -0.4631121 , -1.7533457 , -0.96134007],        [-0.9830793 , -0.1831804 , -1.7907777 , -0.77277637],        [-0.99896413,  0.04550415, -1.7722844 , -0.58851147],       dtype=float32)</code>
Y[0]	
<code>array([-0.97634894,  0.21620078, -1.7363052 , -0.43377334], dtype=float32)</code>	

Do optymalizacji modelu wykorzystaliśmy popularną metodę gradientową Adam, która jest efektywnym algorytmem opartym na metodzie stochastycznego gradientu, odpowiednim dla dużych zbiorów danych. Funkcją kosztu, używaną do minimalizacji błędów predykcji modelu, jest błąd średniokwadratowy (MSE), który dobrze sprawdza się w problemach regresyjnych, takich jak przewidywanie współrzędnych.

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

Podczas trenowania modelu 20% naszych danych zostało wydzielonych jako zbiór walidacyjny, co pozwala na monitorowanie wydajności modelu i unikanie problemu nadmiernego dopasowania (overfittingu).

```
model.fit(X, Y, epochs=50, batch_size=32, validation_split=0.2)
Epoch 50/50
37/37 [=====] - 2s 64ms/step - loss: 0.0013 - val_loss: 0.0025
```

Aby zastosować model do przewidywania, musimy podać N kolejnych iteracji, a model będzie przewidywał następne N iteracji. Proces ten przebiega w podobny sposób, jak w fazie uczenia — dane wejściowe do modelu są ponownie konwertowane na tablice zawierające po 20 zestawów kolejnych klatek (timeframe). Dzięki temu model może przewidywać przyszłe położenia na podstawie wcześniejszych stanów ruchu.

```
X_var = [np.vstack((x1_var[j:j+timeframe], y1_var[j:j+timeframe], x2_var[j:j+timeframe],
                    y2_var[j:j+timeframe])).T for j in range(len(x1_var) - timeframe)]
```

```
predictions = model.predict(X_var)
X_var.shape
```

```
(1481, 20, 4)
```

```
predictions.shape
```

```
(1481, 4)
```

#### 0.5.4. Model przy użyciu RNN oparty na danych uzyskanych numerycznie, nauczony na współrzędnych i prędkościach

Model tworzymy w taki sam sposób jak opisany w poprzednim rozdziale, z tą różnicą, że w tym przypadku do procesu uczenia modelu wykorzystujemy nie tylko współrzędne kątowe ramion wahadła, ale także ich prędkości kątowe. Dzięki temu model ma dostęp zarówno do informacji o położeniu, jak i o dynamice ruchu. Wprowadzenie dodatkowego wymiaru (prędkości) jest, aby model lepiej zrozumiał zależności czasowe i zmiany w ruchu wahadła, czyli dokładniej prognozował kolejne pozycje układu w czasie.

```
for i in range(len(x1_new) - timeframe):
    # Stack the next 'timeframe' positions of both masses to serve as the input sequence
    X.append(np.vstack((x1_new[i:i+timeframe], y1_new[i:i+timeframe], x2_new[i:i+timeframe], y2_new[i:i+timeframe],
                        vx1_new[i:i+timeframe], vy1_new[i:i+timeframe], vx2_new[i:i+timeframe], vy2_new[i:i+timeframe])).T)

    # Record the positions of both masses 20 steps ahead from the current timestep to set as the prediction target
    Y.append([x1_new[i+timeframe], y1_new[i+timeframe], x2_new[i+timeframe], y2_new[i+timeframe],
              vx1_new[i+timeframe], vy1_new[i+timeframe], vx2_new[i+timeframe], vy2_new[i+timeframe]])

model = Sequential([
    LSTM(64, input_shape=(timeframe, 8), return_sequences=True), # First LSTM Layer
    LSTM(32), # Second LSTM Layer with 32 units
    Dense(8) # Output Layer for future x1, y1, x2, y2, vx1, vy1, vx2, vy2
])

X_var = [np.vstack((x1_var[j:j+timeframe], y1_var[j:j+timeframe], x2_var[j:j+timeframe],
                    y2_var[j:j+timeframe], vx1_var[j:j+timeframe], vy1_var[j:j+timeframe],
                    vx2_var[j:j+timeframe], vy2_var[j:j+timeframe])).T for j in range(len(x1_var) - timeframe)]
```

```
predictions = model.predict(X_var)
```

### 0.5.5. Model przy użyciu LNN oparty na danych uzyskanych numerycznie

Dysponując kątami i prędkościami kątowymi obliczonymi numerycznie, możemy zastosować te wartości w równaniu (7) w celu obliczenia przyspieszeń.

```
def f_analytical(state, t=0, m1=1, m2=1, l1=1, l2=1, g=9.8):
    t1, t2, w1, w2 = state
    a1 = (l2 / l1) * (m2 / (m1 + m2)) * jnp.cos(t1 - t2)
    a2 = (l1 / l2) * jnp.cos(t1 - t2)
    f1 = -(l2 / l1) * (m2 / (m1 + m2)) * (w2**2) * jnp.sin(t1 - t2) - \
        (g / l1) * jnp.sin(t1)
    f2 = (l1 / l2) * (w1**2) * jnp.sin(t1 - t2) - (g / l2) * jnp.sin(t2)
    g1 = (f1 - a1 * f2) / (1 - a1 * a2)
    g2 = (f2 - a2 * f1) / (1 - a1 * a2)
    return jnp.stack([w1, w2, g1, g2])

t = np.arange(N, dtype=np.float32) # time steps 0 to N
%time x_train = jax.device_get(solve_analytical(x0, t)) # dynami
%time xt_train = jax.device_get(jax.vmap(f_analytical))(x_train)
```

Zbiór testowy uzyskuje się przy zastosowaniu dalszego kroku N.

```
t_test = np.arange(N, 2*N, dtype=np.float32) # time steps N to 2N
%time x_test = jax.device_get(solve_analytical(x0, t_test)) # dyn
%time xt_test = jax.device_get(jax.vmap(f_analytical))(x_test) #
```

Po zgromadzeniu wszystkich niezbędnych danych można przystąpić do konstrukcji sieci.

Jej budowa będzie następująca:

```
init_random_params, nn_forward_fn = stax.serial(
    stax.Dense(128),
    stax.Softplus,
    stax.Dense(128),
    stax.Softplus,
    stax.Dense(1),
)
```

Wyjściem sieci będzie później obliczony pseudolagrangian.

```
def learned_lagrangian(params):
    def lagrangian(q, q_t):
        assert q.shape == (2,)
        state = normalize_dp(jnp.concatenate([q, q_t]))
        return jnp.squeeze(nn_forward_fn(params, state), axis=-1)
    return lagrangian
```

unkcją kosztów jest średni błąd kwadratowy między danymi uzyskanymi z równania (7), a danymi uzyskanymi z równania (12), przy uwzględnieniu wyjściowego pseudolagrangianu.

```
train_loss = loss(params, (x_train, xt_train))

test_loss = loss(params, (x_test, xt_test))
def loss(params, batch, time_step=None):
    state, targets = batch
    if time_step is not None:
        f = partial(equation_of_motion, learned_lagrangian(params))
        preds = jax.vmap(partial(rk4_step, f, t=0.0, h=time_step))(state)
    else:
        preds = jax.vmap(partial(equation_of_motion, learned_lagrangian(params)))(state)
    return jnp.mean((preds - targets) ** 2)

def equation_of_motion(lagrangian, state, t=None):
    q, q_t = jnp.split(state, 2)
    q_tt = (jnp.linalg.pinv(jax.hessian(lagrangian, 1)(q, q_t)) @
            (jax.grad(lagrangian, 0)(q, q_t) - jax.jacobian(jax.jacobian(lagrangian, 1), 0)(q, q_t) @ q_t))
    return jnp.concatenate([q_t, q_tt])
```

Do procesu optymalizacji znów zastosujemy popularną metodę optymalizacji gradientowej,

jaką jest adam.

```
opt_init, opt_update, get_params = optimizers.adam()
    lambda t: jnp.select([t < batch_size*(num_batches//3),
                          t < batch_size*(2*num_batches//3),
                          t > batch_size*(2*num_batches//3)],
                        [1e-3, 3e-4, 1e-4])
opt_state = opt_init(init_params)
```

Po ustaleniu numeru i liczby batchy można przystąpić do uczenia modelu.

```
batch_size = 100
test_every = 10
num_batches = 500

for iteration in range(batch_size*num_batches + 1):
    if iteration % batch_size == 0:
        params = get_params(opt_state)
        train_loss = loss(params, (x_train, xt_train))
        train_losses.append(train_loss)
        test_loss = loss(params, (x_test, xt_test))
        test_losses.append(test_loss)
        if iteration % (batch_size*test_every) == 0:
            print(f"iteration={iteration}, train_loss={train_loss:.6f}, test_loss={test_loss:.6f}")
        opt_state = update_derivative(iteration, opt_state, (x_train, xt_train))

params = get_params(opt_state)

iteration=50000, train_loss=0.094098, test_loss=0.099857
```

Aby skorzystać z modelu, wystarczy podać współrzędne początkowe oraz pożądany krok czasowy. Model wykorzystuje obliczony pseudolagrangian do przewidywania następnych kroków, korzystając z równania (12).

```
x1 = np.array([3*np.pi/7, 3*np.pi/4, 0, 0], dtype=np.float32)

t2 = np.linspace(0, 20, num=301)

x1_model = jax.device_get(solve_lagrangian(learned_lagrangian(params), x1, t=t2))

def solve_lagrangian(lagrangian, initial_state, **kwargs):
    # We currently run odeint on CPUs only, because its cost is dominated by
    # control flow, which is slow on GPUs.
    @partial(jax.jit, backend='cpu')
    def f(initial_state):
        return odeint(partial(equation_of_motion, lagrangian),
                      initial_state, **kwargs)
    return f(initial_state)
```



---

## Bibliografia

- [1] Charu C. Aggarwal. *Neural Networks and Deep Learning - A Textbook*. Springer, 2018.
- [2] Aurelien Geéron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Sebastopol, CA, 2017.
- [3] L. D. Landau and E. M. Lifshitz. *Mechanics, Third Edition: Volume 1 (Course of Theoretical Physics)*. Butterworth-Heinemann, 3 edition, January 1976.
- [4] P. Mann. *Lagrangian and Hamiltonian Dynamics*. Oxford University Press, 2018.
- [5] T. Nield and G. Werner. *Podstawy matematyki w data science: algebra liniowa, rachunek prawdopodobieństwa i statystyka*. Helion.
- [6] Michael A. Nielsen. Neural networks and deep learning, 2018.



---

## Spis rysunków

1	Rachunek wariacyjny.	3
2	Biologiczne sieci neuronowe.	9
3	Sztuczne sieci neuronowe.	9
4	Reprezentacja etykietyzacji obrazu marchewki za pomocą sieci neuronowych.	10
5	Podstawowa architektura perceptronu.	11
6	Podstawowa architektura sieci feed-forward z dwiema warstwami ukrytymi i jedną warstwą wyjściową.	12
7	Notacja skalarna oraz podstawowa architektura sieci feed-forward z dwiema warstwami ukrytymi i jedną warstwą wyjściową.	12
8	Wykres funkcji signum.	14
9	Wykres funkcji sigmoidalnej.	14
10	Wykres funkcji tgh.	15
11	Wykres funkcji ReLU	15
12	Neuron rekurencyjny (po lewej) rozwinięty w czasie (po prawej)	19
13	Warstwa neuronów rekurencyjnych (po lewej) rozwinięta w czasie (po prawej).	20
14	Stan ukryty komórki i jej wyjście mogą być różne.	21
15	Komórka LSTM	22