Politechnika Wrocławska
Wydział Elektroniki, Fotoniki i Mikrosystemów

KIERUNEK: Automatyka i Robotyka

PRACA DYPLOMOWA INŻYNIERSKA

TYTUŁ PRACY:

Sztuczna inteligencja w roli gracza w grze zręcznościowej

Artificial intelligence as a player in an arcade game

AUTOR:
Jan Bronicki

PROMOTOR:
Dr inz. Mariusz Uchroński

WROCŁAW 2022

Spis treści

Skróty

HTML (ang. *HyperText Markup Language*)
JSON (ang. *JavaScript Object Notation*)
JIT (ang. *Just In Time compiler*)
PyPI (ang. *Python Package Index*)
NumPy (ang. *Numerical Python*)
SciPy (ang. *Scientific Python*)
SymPy (ang. *Symbolic Python*)
API (ang. *Application Programming Interface*)
GUI (ang. *Graphical User Interface*)
GTK (ang. *GIMP ToolKit*)
PEP (ang. *Python Enhancement Proposal*)
PIP (ang. *Preferred Installer Program*)
POSIX (ang. *The Portable Operating System Interface*)
DLL (ang. *Dynamic-Link Library*)
CI (ang. *Continuous Integration*)
CD (ang. *Continuous Delivery*)

Z dedykacją, dla Łukasza Stanisławowskiego

Rozdział 1

Wstęp

Rozdział 2

Cel pracy

Celem pracy jest stworzenie sieci neuronowej zdolnej do samodzielnej nauki gry w dowolną grę zręcznościową. Model sieci neuronowej będzie oparty o uczenie przez wzmacnianie w połączeniu z Q-learning'iem. Zdecydowano, że sieć zostanie napisana w języku Python, przy pomocy biblioteki PyTorch. W celu dystrybucji sieci neuronowej do użytku, dla różnych gier stworzona zostanie paczka języka Python, przy pomocy narzędzia Cookiecutter, która zostanie opublikowana na oficjalnym indeksie PyPI. W celu przetestowania paczki oraz sieci neuronowej zostanie stworzona gra w stylu Snake, przy użyciu biblioteki PyGame w celu wizualizacji rozgrywki.

Układ pracy jest następujący. Rozdział 3 przedstawia użyte narzędzia oraz technologie używane przy budowie sieci neuronowej, paczki Python, oraz gry w stylu Snake służącej do testów. Rozdział 4 omawia czym są sieci neuronowe, czym jest model uczenia przez wzmacnianie w połączeniu z Q-learningiem, oraz jego implementację w projekcie. W rozdziale 5 omówiony zostaje system paczek języka Python, oraz jego system indeksowania, narzędzia takie jak Cookiecutter służące do prostego tworzenia paczek z szablonów, następnie proces tworzenia, opublikowania i zainstalowania utworzonej paczki. W 6 rozdziale zaprezentowane zostanie użycie paczki z siecią neuronową na stworzonej, dla przykładu grze typu Snake. W rozdziale 7 zostanie podsumowana cała praca inżynierska. Zostaną tam wyciągnięte wnioski oraz spostrzeżenia powstałe podczas tworzenia projektu oraz przedstawione zostaną możliwości rozwinięcia projektu.

Rozdział 3

Użyte narzędzia i technologie

3.1. Język Python

Język Python [32] [42] [35] jest bardzo popularnym, nowoczesnym oraz wysokopoziomowym językiem programowania. Czytając artykuły i inne treści na temat historii Python'a [45] [41] dowiadujemy się, że język powstał w 1991 roku. Stworzony przez Guido van Rossum podczas swojej pracy w laboratorium w Centrum Matematyki i Informatyki w Amsterdamie, pierwotnie tworzony był z myślą zastąpienia rozwijanego w latach osiemdziesiątych języka ABC. Samą nazwę język Python nazwę zawdzięcza popularnemu serialowi komediowemu emitowanemu przez BBC w latach siedemdziesiątych - "Latający Cyrk Monty Pythona", którego Guido był fanem. Projekt początkowo zakładał stworzenie prostego w użyciu, zwięzłego i wysokopoziomowego języka, głównie na potrzeby pracy w Amsterdamskim laboratorium. Z biegiem czasu Python stał się rozwijanym przez społeczność programistyczną projektem Open Source, nad którym czuwała organizacja non-profit założona przez Guido von Rossum'a, Python Software Foundation.

Python jest multi-paradygmatowym językiem, gdzie programowanie obiektowe i strukturalne są w pełni wspierane, wiele cech języka wspiera programowanie funkcyjne i aspektowe. Wiele innych paradygmatów jest wspieranych dzięki modularnym rozszerzeniom do języka. Python w przeciwieństwie do języków statycznie typowanych takich jak C, Rust czy TypeScript, stosuje typowanie dynamiczne, które sprawdza poprawność i bezpieczeństwo typów w programie, dynamicznie, podczas jego egzekucji, w przeciwieństwie do typowania statycznego, gdzie typy sprawdzane są podczas kompilacji kodu. Dodatkowo Python posiada kombinacje zliczania referencji oraz cyklicznego Garbage Collector'a. Architektura języka Python oferuje wsparcie, dla programowanie funkcyjnego zgodnego z tradycją języka Lisp. Język posiada typowe, dla języków funkcyjnych funkcje takie jak `filter`, `map`, `reduce`, `lambda`, list comprehensions, słowniki oraz generatory. Standardowa biblioteka języka posiada moduły `itertools` oraz `functools`, które implementują narzędzia funkcjonalne zapożyczone z języków Haskell oraz Standard ML.

Zamiast polegać na wbudowanej funkcjonalności w jądrze języka, Python został zaprojektowany tak, aby być najmożliwiej elastyczny, aby mógł współdziałać z różnymi odrębnymi modułami. Sama kompaktowa modularność sprawiła, że język stał się popularnym sposobem, na dodawanie programowalnego interfejsu to istniejących już aplikacji oraz języków programowania. CPython jest referencyjną implementacją Python'a, napisaną w C, która spełnia standard C89 z niektórymi cechami standardu C99. Jedną z charakterystyk

implementacji CPython'a jest to, że zespół odpowiedzialny za rozwój oraz utrzymanie kodu jego interpretera preferuje odrzucanie poprawek do kodu mających na celu marginalną poprawę szybkości i sprawności interpretera w zamian za zachowanie czystości i czytelności kodu źródłowego. Taka długoterminowa postawa umożliwiła powstanie powstanie innych implementacji języka Python opartych o inne rozwiązania, bądź inne języki za pomocą których napisany został sam interpreter. Dzięki różnorodności implementacji Python'a, jego modularności i łatwej rozbudowie o obce rozszerzenia programista może zdecydować się na przeniesienie wrażliwych na czas wykonywania funkcjonalności do odrębnych modułów napisanych w innych językach takich jak C, lub Rust, albo użyć wyspecjalizowanej do tego odmiany samego interpretera takiej jak PyPy, która posiada tak zwany JIT, czyli Just-In-Time compiler, pozwalający na optymalizację kodu na system bądź architekturę docelową danego programu.

Dzięki tak rozbudowanemu środowisku oraz społeczności otaczającej język powstała ogromna biblioteka paczek, dla języka Python, która często podawana za jedną z największych zalet języka. W ten sposób Python stał się swoistym odpowiednikiem *Lingua Franca* wśród programistów. Z powodu wyżej wymienionych cech i okoliczności Python jest wykorzystywany w bardzo różniących się, oraz na pozór nie połączonych ze sobą dziedzinach takich jak Web Development, Automatyzacja, Bazy Danych, Aplikacje Mobilne, Testowanie Oprogramowania, Analiza Danych oraz Uczenie Maszynowe.

3.2. Biblioteka PyTorch

PyTorch [18] [33] [38] [37] jest Open Source'ową biblioteką służącą do uczenia maszynowego. Jest bazowana na bibliotece Torch napisanej w języku Lua. Z powodu niszowości języka Lua, oraz braku modularności i możliwości rozbudowywania go o nowe funkcjonalności za pomocą zewnętrznych modułów i paczek, powstał PyTorch, czyli, biblioteka Torch, ale zaimplementowana w języku Python 3.1. Dzięki temu PyTorch może korzystać z bardzo rozbudowanego środowiska Python, które oferuje dużą ilość naukowych paczek, między innymi takich jak NumPy 3.3.

Jedną z największych zalet biblioteki PyTorch jest możliwość programowanie imperatywnego. Jest to przeciwieństwem do bibliotek takich jak TensorFlow i Keras, które ze względu na poleganie głównie na językach takich jak C i C++, oferują jedynie możliwość programowania symbolicznego. Większość Python'owego kodu jest imperatywne jako, że jest to dynamicznie interpretowany język. W sytuacji symbolicznej zachodzi przeciwieństwo, ponieważ istnieje bardzo wyraźne rozróżnienie pomiędzy zdefiniowaniem grafu komputacyjnego, a jego kompilacją. W przypadku imperatywnym komputacja zachodzi w momencie jej wywołania, nie we wcześniej zoptymalizowanym punkcie w kodzie. Podejście symboliczne pozwala na większą optymalizację, a imperatywne takie jak PyTorch pozwalaj na większą swobodność, oraz używanie natywnych cech, funkcjonalności i rozszerzających modułów języka Python. Drugą największą zaletą biblioteki PyTorch są dynamiczne grafy komputacyjne, które w przeciwieństwie do bibliotek takich jak TensorFlow generują je statycznie przed uruchomieniem programu. PyTorch umie generować i modyfikować je dynamicznie podczas działania programu.

3.3. Biblioteka NumPy

NumPy [34] [26] [13] [12] jest Open Source biblioteką stworzoną, dla języka programowania Python 3.1, dodaje wsparcie, dla dużych wielowymiarowych tablic i macierzy wraz z dużą kolekcją wysokopoziomowych funkcji matematycznych, które pozwalają operować na wspomnianych macierzach. Poprzednikiem biblioteki NumPy był, Numeric, oryginalnie stworzony przez Jim'a Hugunin'a wraz z kontrybucjami kilku innych developerów. W roku 2005, Travis Oliphant stworzył projekt NumPy włączając w to właściwości oraz funkcjonalności Numeric'a. Python nie był oryginalnie stworzony do numerycznej komputacji, ale już we wczesnym życiu języka różne towarzystwa naukowe i inżynieryjne wyrażały swoje zainteresowanie językiem. NumPy adresuje problem powolności języka Python poprzez zapewnienie wielowymiarowych macierzy, funkcji i operacji, które są wydajne obliczeniowo operując na macierzach.

Używanie biblioteki NumPy w Python'ie funkcjonalnością przypomina programowanie w środowisku MATLAB, jako, że oba są interpretowane, mają podobną składnie, oba pozwalają użytkownikowi pisać szybkie i wydaje programy tak długo jak operacje przeprowadzane są na macierzach. W przeciwieństwie do MATLAB'a, NumPy nie oferuje tak wielkiej ilości dodatkowych narzędzi. Oferuje odwrotne podejście, gdzie to inne paczki/narzędzia korzystają u swoich podstaw z biblioteki NumPy. Dzięki zaawansowanej integracji z Python'em oraz byciu podłożem, dla zasadniczej większości paczek i narzędzi służących celom obliczeniowo naukowym takim jak SciPy, SymPy, Scikit-Learn i wielu innym NumPy, stał się podstawową warstwą, dla takich narzędzi a tym samym umożliwia im prostą komunikację między sobą jako, że macierze na których operują są macierzami biblioteki NumPy.

Główną funkcjonalnością biblioteki NumPy jest jej `ndarray`, struktura danych, reprezentująca n-wymiarową macierz. Wewnętrzna niskopoziomowa implementacja owych macierzy polega na kroczących widokach w pamięci. Takowe dane muszą być homogenicznego typu. Takie widoki pamięci mogą być również bufferami zaalokowanymi z poziomu innych języków takich jak C, C++, czy Fortran. Powoduje to ogromną optymalizację, jako, że eliminuje to potrzebę kopiowania i przenoszenia danych.

3.4. Biblioteka Matplotlib

Biblioteka Matplotlib [25] [39] [34] [8] jest najpopularniejszym Python'owym 3.1 narzędziem służącym do tworzenia wykresów, grafów, histogramów, obrazów, wielowymiarowych grafów i rysunków służących do wizualizacji danych. Oryginalnie stworzona przez John D. Hunter'a miała za zadanie tworzyć przyzwoicie wyglądające wykresy, które można by poddać publikacji. Mimo, że inne biblioteki są dostępne większość programistów używa Matplotlib'a jako, że jest on najpopularniejszy oraz najbardziej i najlepiej rozbudowany ze wszystkich nie wspominając o jego wpasowaniu w istniejący ekosystem Data Science.

Matplotlib zapewnia obiektowo zorientowane API do tworzenia i używania generowanych wykresów w toolkitach GUI takich jak Tkinter wxPython, Qt, lub GTK. Dodatkowo istnieje również proceduralny interfejs "pylab" bazujący na maszynie stanu (podobnie do OpenGL), zaprojektowany, aby przypominać interface MATLAB'a.

Pyplot to moduł Matplotlib'a, który jest zaprojektowany z myślą bycia używalnym pod kątem programistycznym tak jak MATLAB, ale z dodatkową zaletą pozostawania w przestrzeni języka Python, która jest Open Source i darmowa.

3.5. PyGame

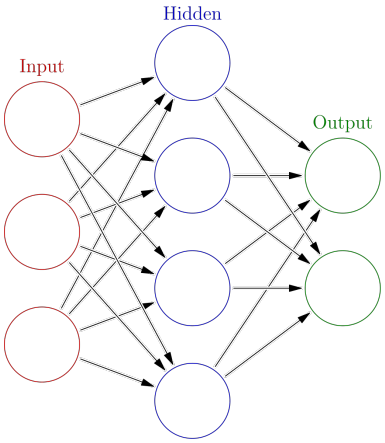
PyGame [16] [15] jest cross-platform'ową biblioteką stworzoną, dla języka Python 3.1 zaprojektowaną z myślą tworzenia prostych gier i animacji. Zawiera w sobie biblioteki odpowiadające za grafikę i dźwięk. PyGame, był oryginalnie napisany przez Pete Shinners'a, w celu zastąpienia PySDL po tym jak jego developemnt został zatrzymany. Od roku 2000 jest to projekt społeczności i został objęty licencją GNU Lesser General Public License, która pozwala na dystrybucje PyGame'a zarówno wraz z Open Source'owym oprogramowaniem jak i tym prawnie zastrzeżonym prawami autorskimi.

Rozdział 4

Sieć neuronowa

4.1. Sieć neuronowa

Sieć neuronowa [10] [2] [40] [29] jest siecią bądź układem neuronów, w nowoczesnym zrozumieniu pojęcie jest stosowane głównie w odniesieniu do sztucznych sieci neuronowych , które składają się ze sztucznych komputerowo symulowanych, uproszczonych modeli neuronów. Sztuczne sieci neuronowe są inspirowane przez biologiczne sieci neuronowe, które tworzą mózgi zwierząt. Każde połączenie, czyli inaczej każda synapsa w biologicznej sieci neuronowej może transmitować sygnał do innych neuronów. Sztuczny neuron otrzymuje sygnał, następnie przetwarza go i może zasygnalizować inne połączone z nim neurony. Sam sygnał jest liczbą rzeczywistą, wyjście każdego neuronu obliczane za pomocą jakiejś nie liniowej sumy wejścia. Połączenia neuronów posiadają wagę, która jest dopasowywana podczas procesu uczenia. Waga danego połączenia zwiększa lub pomniejsza siłę sygnału w danym połączeniu. Neurony mogą posiadać granice wartości takiego sygnału, taką, że dany sygnał zostanie przepuszczony jeśli zsumowany sygnał przekracza jakąś wartość. Zazwyczaj neurony są agregowane w tak zwane warstwy. Różne warstwy mogą powodować różne transformacje sygnału bazując na ich wejściach. Sygnał podróżuje od warstwy wejściowej, do ostatniej warstwy, która jest warstwą wejściową, zwykle po przejściu kilku wewnętrznych warstw. Poniższy rysunek przedstawia symboliczny graf sieci neuronowej 4.1.



Rys. 4.1: Symboliczny model sieci neuronowej [6]

W 1943 roku Warren McCulloch i Walter Pitts zapoczątkowali dziedzinę poprzez stworzenie stworzenie obliczeniowego modelu sieci neuronowej. W późnych latach 40, dwudziestego wieku, D. O. Hebb stworzył hipotezę bazującą na mechanizmie neuroplastyczności [11], czyli zdolności tkanki nerwowej do tworzenia nowych połączeń mających na celu jej reorganizację, adaptację lub zmianę. Jest to powszechna cecha neuronów, która występuje na każdym poziomie układu nerwowego. Teoria Hebbiana postuluje zwiększenie wydajności synaps na skutek ciągłej stymulacji. W roku 1945 Farley i Wesley A. Clark jako pierwsi użyli maszyny obliczeniowej (wtedy nazywanej jeszcze "kalkulatorami"), do symulacji sieci neuronów Hebbiana. Pierwsze funkcjonalne sieci neuronowe z wieloma warstwami zostały opublikowane przez Ivakhnenko i Lape w roku 1965. A podstawy algorytmu propagacji wstecznej (z ang. backpropagation) w roku 1960 przez Kelly'iego, oraz Bryson'a w 1961.

W uczeniu maszynowym propagacja wsteczna [3] jest szeroko używanym algorytmem służącym do trenowania sieci neuronowych. Przy szkoleniu sieci neuronowej propagacja wsteczna oblicza gradient funkcji błędu względem wagi sieci, dla konkretnego wejścia/wyjścia, i robi to efektywnie pod względem obliczeniowym w przeciwieństwie do naiwnych metod obliczeniowych obliczania gradientu względem każdej wagi indywidualnie. Efektywność sprawia, że możliwe jest użycie metod gradientowych do trenowania sieci wielowarstwowych, uaktualniania wag, aby zminimalizować funkcję błędu. Metoda gradientu prostego, lub jego różne warianty takie jak metoda gradientu stochastycznego są bardzo często używane. Algorytm propagacji wstecznej działa na zasadzie obliczania gradientu funkcji błędu każdej z wagi korzystając z reguły łańcuchowej, obliczając gradient każdej warstwy po kolei, iterując w tył od ostatniej warstwy, aby uniknąć powtarzających się obliczeń, co między innymi jest przykładem programowania dynamicznego.

Początek sztucznych sieci neuronowych był spowodowany inspiracją i próbą użycia architektury ludzkiego mózgu tak, aby mógł on wykonywać zadanie, które są bardzo problematyczne, dla konwencjonalnych algorytmów. Dość szybko nastąpiła reorientacja ku ulepszeniu empirycznych wyników, opuszczając tym samym próby prawdziwego, zgodnego z ich biologicznymi prekursorami. Neurony połączone są ze sobą różnorakimi metodami tak, aby umożliwić wyjście niektórych neuronów, aby stało się wejściem innych. W ten sposób sieć tworzy ukierunkowany graf wag.

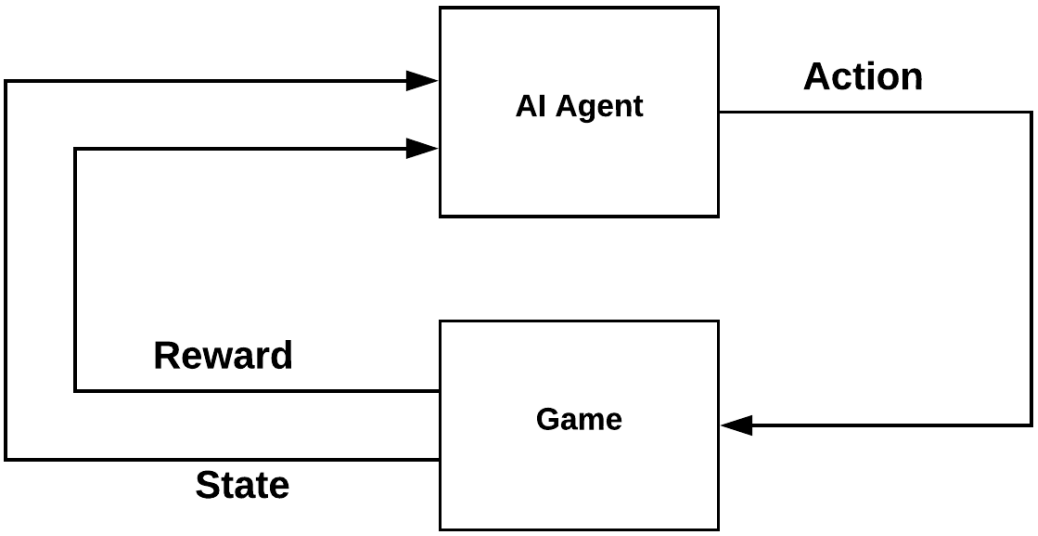
Sztuczna sieć neuronowa składa się z kolekcji symulowanych neuronów. Każdy neuron jest wierzchołkiem w grafie, który jest połączony z innymi wierzchołkami za pomocą połączeń nazywanych krawędziami. Każda krawędź ma wagę, która determinuje siłę z jaką oddziaływuje na inne neurony.

4.2. Uczenie przez wzmacnianie

Uczenie przez wzmacnianie (ang. reinforcement learning, RL) [22] [43] [31] jest jednym z typów uczenia maszynowego, które koncentruje się na inteligentnych agentach, które dokonują akcji w danych środowiskach tak, aby zmaksymalizować kumulacyjną nagrodę. Reinforcement learning jest jednym z trzech podstawowych paradygmatów uczenia maszynowego, obok uczenia nadzorowanego (Supervised Learning), oraz uczenia nienadzorowanego (Unsupervised Learning).

Reinforcement learning różni się od uczenia nadzorowanego tym, że nie potrzebuje poukładanego, oznakowanego zestawu danych wejść i wyjść, które mają zostać zaprezentowane sieci. Dodatkowo sub-optymalne zachowania i rozwiązania sieci nie wymagają korekcji.

Zamiast tego trening sieci skupia się na balansie pomiędzy eksploracją nowych zachowań, oraz eksploatacją już istniejących umiejętności. Interakcja pomiędzy siecią, a środowiskiem przebiega w następujący sposób, gdzie to agent umieszczony w danym środowisku sczytuje jego stan. Stan środowiska jest wejściem sieci neuronowej. Następnie na podstawie danego stanu podejmuje decyzję, która jest wskazana przez wyjście sieci neuronowej. Podjęta decyzja wpływa na środowisko, następnie na podstawie rezultatu uzyskanego poprzez daną akcję dokonywana jest poprawa wag połączeń sieci. Poniższy diagram 4.2 przedstawia uproszczony schemat tego działania, biorąc za środowisko grę komputerową.



Rys. 4.2: Symboliczny model sieci w połączeniu z grą

4.3. Q-learning

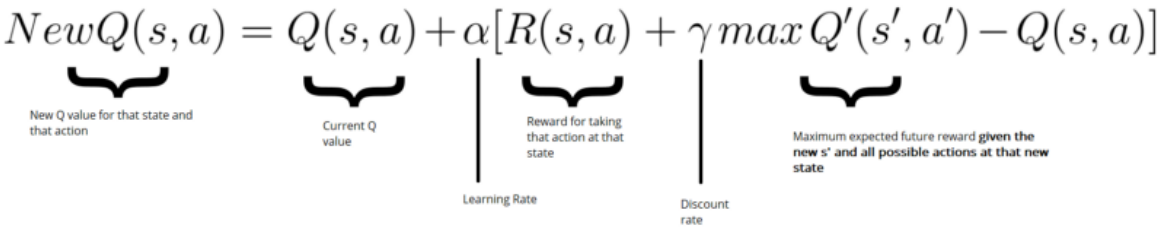
Q-learning [30] [20] jest to bezmodelowy algorytm nauczania przez wzmacnianie, którego zadaniem jest nauczenie się wartości danej akcji w danym środowisku i stanie. Nie wymaga modelu środowiska. Ważną wartością w tej metodzie nauczania jest wartość Q, która oznacza jakość wykonanej akcji. Implementacja Q-learningu zakłada następującą procedurę:

1. Inicjalizacja wartości Q
2. Wybranie akcji przez model
3. Wykonanie akcji
4. Pomiar nagrody
5. Zaktualizowanie wartości Q + trening modelu
6. Powrót do punktu numer 1.

W pierwszym kroku inicjalizujemy wartość Q, poprzez inicjalizację naszego modelu losowymi parametrami. Następnie w drugim kroku wybieramy akcję, którą ma wykonać nasz agent. Akcja jest wybierana na podstawie predykcji modelu wykonując metodę

`model.predict(state)`, dodatkowo co jakiś arbitralny procent ruchów decydujemy się na losowy ruch w celu umożliwienia modelowi eksploracji innych możliwości niż te już potencjalnie wyuczone, co jest głównie przydatne na początku nauki modelu, kiedy nie ma on doświadczenia ze środowiskiem. Kolejnym krokiem jest wykonanie danej akcji na środowisku i zmierzenie nagrody, która została spowodowana danym ruchem, dzięki tej zmierzonej wartości możliwe jest dokonanie aktualizacji wartości Q, która jest obiektywnym miernikiem jakości akcji, dzięki czemu możliwe jest trenowanie neuronów na danej akcji.

Funkcją błędu odpowiedzialną za obliczenie wartości Q jest równanie Bellman'a 4.3 [4], gdzie nowa wartość Q jest obliczana z sumy starej wartości Q z współczynnikiem uczenia pomnożonym przez sumę nagrody za dokonaną akcję, wraz z maksymalną spodziewaną przyszłościową wartością Q biorąc pod uwagę nowy stan przemnożoną przez discount-rate minus obecna wartość Q.



Rys. 4.3: Równanie Bellman'a [23]

Dzięki równaniu Bellman'a możliwe jest skonstruowanie nowej funkcji błędu średniokwadratowego [5] opartego o wartość Q 4.1.

$$Błąd = (Q_{nowe} - Q)^2 \tag{4.1}$$

4.4. Implementacja

Import potrzebnych bibliotek 4.1 takich jak NumPy 3.3, PyTorch 3.2, Matplotlib 3.4:

Listing 4.1: Importowanie bibliotek i ich modułów

```
import os
import random
from abc import ABC, abstractmethod
from typing import Tuple

import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import matplotlib.pyplot as plt
from IPython import display
```

Na początku definiowane jest kilka podstawowych parametrów 4.3 takich jak maksymalna liczba trzymanych obiektów w pamięci, batch-size [36] (czyli liczbę egzemplarzy treningowych na których będzie szkolona sieć naraz), oraz współczynnik uczenia, czyli parametr wiążący funkcję błędu ze zmianą wag.

Listing 4.2: Podstawowe parametry

```
MAX_MEMORY = 100000
BATCH_SIZE = 1000
LEARNING_RATE = 0.001
```

Kolejnym krokiem jest definicja klasy `Agent` 4.3, która definiuje parametry takie jak rozmiar warstw wejściowych, wewnętrznych/ukrytych, oraz wyjściowych sieci neuronowej, współczynnik ε kontrolujący losowość akcji agenta, oraz współczynnik gamma, inaczej zwany jako discount-rate. Dodatkowo tworzymy model sieci neuronowej `self.model` korzystając z klasy `LinearQNet` z biblioteki PyTorch, oraz `self.trainer`, który jest obiektem odpowiedzialnym za trening sieci neuronowej, zgodnie z podanymi parametrami.

Listing 4.3: Definicja klasy `Agent`

```
class Agent:
    def __init__(
        self,
        input_size: int,
        output_size: int,
        epsilon: float = 0,
        gamma: float = 0.9,
        hidden_size: int = 256,
    ) -> None:
        self.number_of_games = 0
        self.epsilon = epsilon  # Controls randomness
        self.gamma = gamma  # Discount rate
        self.memory = deque(maxlen=MAX_MEMORY)
        self.model = LinearQNet(input_size, hidden_size, output_size)
        self.trainer = QTrainer(
            self.model, learning_rate=LEARNING_RATE, gamma=self.gamma
        )
```

Utworzona klasa posiada metodę `remember()` 4.4, służącą do dodania stanu, akcji, nagrody, stanu będącego rezultatem akcji, oraz stanu gry, do specjalnej listy, która je zachowuje w celu przeprowadzenia treningu sieci neuronowej.

Listing 4.4: Definicja metody `remember`

```
def remember(self, state, action, reward, next_state, game_state):
    self.memory.append((state, action, reward, next_state, game_state))
```

Metoda `get_action()` 4.5 zwraca podjętą akcję przez model. Z uwagi na balans pomiędzy eksploracją, a eksploatacją w uczeniu wzmacnianym we wczesnych fazach gry implementujemy możliwość losowych ruchów tak, aby agent miał możliwość eksploracji.

Listing 4.5: Definicja metody `get_action`

```
def get_action(self, state):
    # Get random moves
    self.epsilon = 80 - self.number_of_games
    final_move = [0, 0, 0]
    if random.randint(0, 200) < self.epsilon:
        move = random.randint(0, 2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float)
        prediction = self.model(state0)
        move = torch.argmax(prediction).item()
        final_move[move] = 1

    return final_move
```

Następnie definiowane są metody nauki krótko pamięciowej, długo pamięciowej, które wzywają metodę `self.trainer.train_step()` klasy `QTrainer`. Metoda `self.train_long_memory()` 4.6 jest odpowiedzialna za trenowanie sieci neuronowej na batch'u wyników gry po jej zakończeniu. W ten sposób tworzona jest długoterminowa pamięć agenta, która jest w stanie uczyć się z błędów oraz sukcesów całej rozgrywki.

Listing 4.6: Metoda `train_long_memory()`

```
def train_long_memory(self):
    if len(self.memory) > BATCH_SIZE:
        mini_sample: List[Tuple] = random.sample(self.memory, BATCH_SIZE)
    else:
        mini_sample: List[Tuple] = self.memory
    states, actions, rewards, next_states, game_states = zip(*mini_sample)
    self.trainer.train_step(states, actions, rewards, next_states, game_states)
```

Metoda `self.train_short_memory()` 4.7 jest odpowiedzialna za trenowanie sieci neuronowej przy każdym ruchu w grze, dzięki czemu tworzona jest pamięć krótkoterminowa i agent jest w stanie uczyć się na świeżo popełnionych błędach, oraz sukcesach w rozgrywce.

Listing 4.7: Metoda `train_short_memory()`

```
def train_short_memory(self, state, action, reward, next_state, game_state):
    self.trainer.train_step(state, action, reward, next_state, game_state)
```

Klasa `QTrainer` 4.8 jest odpowiedzialna za trenowanie sieci neuronowej używając wskazanych parametrów takich jak współczynnik nauczania (learning rate), czy współczynnik gamma.

Listing 4.8: Definicja klasy `QTrainer`

```
class QTrainer:
    def __init__(self, model: nn.Module, learning_rate, gamma) -> None:
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=learning_rate)
        self.criterion = nn.MSELoss()
```

Adam (`optim.Adam()`) [27] [19] jest algorytmem optymalizacyjnym, które jest rozwinięciem metody gradientu prostego, która została niedawno szeroko zaadaptowana, dla algorytmów trenowania sztucznej inteligencji. Jako funkcję błędu ustawiony zostaje błąd średniokwadratowy z biblioteki PyTorch `nn.MSELoss()`.

Metoda `self.trainer.train_step()` 4.9 jest zaimplementowana wewnątrz klasy `QTrainer`, odpowiada ona za przeprowadzenie każdego kroku nauki sieci neuronowej. Konwertujemy zmienne reprezentujące stan, nowy stan po akcji agenta, akcje, oraz nagrodę do typów tensora z biblioteki PyTorch. Z uwagi na możliwość przekazania parametrów do nauki w formie batch'a, odpowiednik warunek konwertuje je na poprawne tensory. Kolejnym krokiem jest pozyskanie wartości parametru Q, poprzez predykcję modelu dzięki czemu jesteśmy w stanie zaktualizować wartość parametru Q, następnie przechodzimy do faktycznego treningu naszego modelu. Za pomocą metod biblioteki PyTorch obliczamy gradient, funkcję błędu, i propagację wsteczną, dla modelu.

Listing 4.9: Metoda `train_step()` klasy `QTrainer`

```
def train_step(self, state, action, reward, next_state, game_state):
    state = torch.tensor(state, dtype=torch.float)
    next_state = torch.tensor(next_state, dtype=torch.float)
    action = torch.tensor(action, dtype=torch.long)
    reward = torch.tensor(reward, dtype=torch.float)

    if len(state.shape) == 1:
        state = torch.unsqueeze(state, 0)
        next_state = torch.unsqueeze(next_state, 0)
        action = torch.unsqueeze(action, 0)
        reward = torch.unsqueeze(reward, 0)
        game_state = (game_state,)

    # Predicted Q values with current state
    prediction = self.model(state)
    target = prediction.clone()
    for each_game_state in range(len(game_state)):
        Q_new = reward[each_game_state]
        if not game_state[each_game_state]:
            Q_new = reward[each_game_state] + self.gamma * torch.max(
                self.model(next_state[each_game_state])
            )

        target[each_game_state][
            torch.argmax(action[each_game_state]).item()
        ] = Q_new

    # Q new = r + y * max(prediction Q value)
    self.optimizer.zero_grad()
    loss = self.criterion(target, prediction)
    loss.backward()

    self.optimizer.step()
```

Klasa `Trainer` 4.10 ma za zadanie połączyć ze sobą grę na, której agent ma się uczyć wraz z modelem sieci neuronowej. Do tego celu została stworzona klasa, która dziedziczy ze standardowej biblioteki ABC (Abstract Classes). Pozwala to na użycie klasy `Trainer` jako definicji interfejsu dzięki, któremu gra mogłaby się połączyć z modelem.

Listing 4.10: Definicja klasy `Trainer`

```
class Trainer(ABC):
    def __init__(self, game, input_size: int, output_size: int, hidden_size: int):
        self.game = game
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
```

Sama klasa definiuje rozmiary sieci neuronowej, ale to jej abstracyjne metody (`get_state()` i `perform_action()` 4.11), czyli inaczej funkcje, które klasy dziedziczące z klasy `Trainer` muszą zaimplementować definiują w jaki sposób siec neuronowa będzie dostawała informacje ze środowiska, oraz w jaki sposób będzie na nie działać swoimi akcjami.

Listing 4.11: Abstrakcyjne metody służące jako definicja interfejsu

```
@abstractmethod
def get_state(self) -> np.ndarray:
    pass

@abstractmethod
def perform_action(self) -> Tuple[int, bool, int]:
    pass
```

Klasa `Trainer` posiada metodę `train()` 4.12, która jest odpowiedzialna za uruchomienie całej procedury uczenia maszynowego na zdefiniowanym interfejsie z danym środowiskiem. Parametrem jaki przyjmuje jest nazwa pod jaką ma zostać zapisana sieć neuronowa i jej wagi w postaci binarnej na przyszły użytek użytkownika. W środku funkcji istnieje pętla `while`, która jest odpowiedzialna za proces nauki sieci. Na początku zapisujemy stary stan gry, następnie z obecnego modelu pobieramy akcję. Po zastosowaniu akcji na środowisko zapisujemy rezultat jaki dała akcja, oraz zapisujemy stan gry po przeprowadzaniu akcji. Następnie trenujemy pamięć krótką na wykonanej akcji, oraz zapisujemy daną akcję i jej rezultaty, na późniejszy trening pamięci długotrwałej. Następnie sprawdzany jest stan gry. Jeśli gra dobiegła końca trenowana jest pamięć długotrwała na całej rozgrywce, wytrenowany model jest zapisywany do wskazanego pliku, oraz wyrysowywany jest graf wyników.

Listing 4.12: Implementacja metody `train()`

```
def train(self, model_file: str = "model.pth"):
    plot_scores = []
    plot_mean_scores = []
    total_score = 0
    record = 0
    agent = Agent(
        input_size=self.input_size,
        output_size=self.output_size,
        hidden_size=self.hidden_size,
    )
    while True:
        # Get the old state
        old_state = self.get_state()
        # Get the move based on the State
        final_move = agent.get_action(old_state)
        # Perform the Move and get the new State
        reward, game_over, game_score = self.perform_action(final_move)
        new_state = self.get_state()
        # Train the short memory ON EACH MOVE
        agent.train_short_memory(
            state=old_state,
            action=final_move,
            reward=reward,
            next_state=new_state,
            game_state=game_over,
        )
        # Remember the state for long term training
        agent.remember(
            state=old_state,
            action=final_move,
            reward=reward,
            next_state=new_state,
            game_state=game_over,
        )
        if game_over:
            # Train long memory, plot the result
            self.game.reset_game_state()
            agent.number_of_games += 1
            agent.train_long_memory()
            # Save the newest record
            if game_score > record:
                record = game_score
                agent.model.save_model(file_name=model_file)
            print(
                f"Game {agent.number_of_games}, Score {game_score}, Record {record}"
            )
            # Plot the scores
            plot_scores.append(game_score)
            total_score += game_score
            mean_score = total_score / agent.number_of_games
            plot_mean_scores.append(mean_score)
            plot(plot_scores, plot_mean_scores)
```

Rozdział 5

Paczka Python

5.1. PyPI

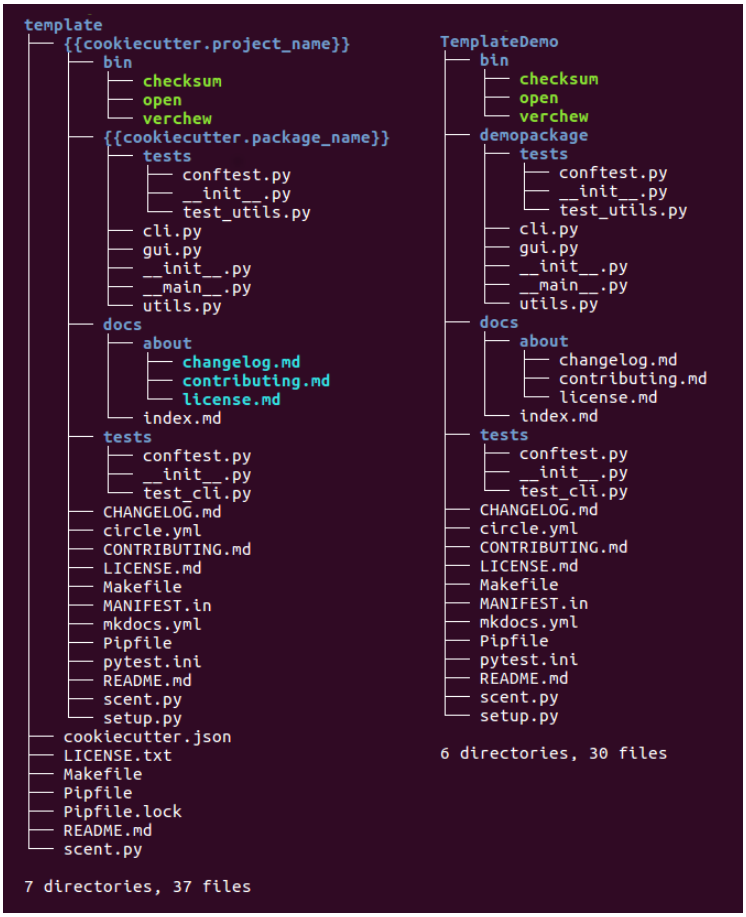
PyPI, [17] [42] czyli Python Package Index jest oficjalnym repozytorium, dla języka Python 3.1 przeznaczonym na third-party paczki przeznaczone, dla języka, które może zainstalować każdy użytkownik. PyPI jest operowany przez Python Software Foundation, która jest non-profit organizacją odpowiedzialną za utrzymywanie języka Python i jego środowiska. Niektóre menadżery paczek takie jak pip [14] używają PyPI jako domyślnego źródła paczek i ich zależności. Na dzień 30 października 2021 roku, ponad $336\ 000$ Python'owych paczek może zostać w ten sposób zainstalowane. PyPI głównie hostuje Python'owe paczki w formie nazywanej "sdists", czyli z *ang. "source distributions*", lub w formie pre-kompilowanej tak zwanej "wheels". PyPI jako indeks paczek pozwala użytkownikowi na wyszukiwanie paczek za pomocą konkretnych słów lub filtrów, przeszukując metadane paczek, takie jak licencje czy kompatybilność ze standardem POSIX. Pojedyncza paczka na PyPI może posiadać, pomijając metadane, poprzednie wersje paczki, pre-kompilowane wheel'e (na przykład zawierające DLL'e na system Windows), lub wiele innych form przewidzianych na inne systemy operacyjne i wersje Python'a.

Python Distribution Utilities, czyli w skrócie nazywane "distutils", to moduł dodany do Python'a, we wrześniu roku 2000, jako część standardowej biblioteki języka w wersji `1.6.1`, 9 lat po wyjściu pierwszej oficjalnej wersji Python'a, mając na celu uproszczenie instalacji third-party paczek. Jednakże, distutils dawało jedynie narzędzie służące do tworzenia paczek, dla kodu napisanego w Python'ie, i nic więcej. Umiało zebrać, i dystrybuować metadane paczki, ale nie używało ich w żaden inny sposób. Python ciągle potrzebował scentralizowanego katalogu paczek znajdującego się w internecie. PEP 241 [1] zaproponował standaryzacje indeksów, która została sfinalizowana w marcu 2001 roku.

5.2. Cookiecutter

Cookiecutter [7] [21] to command-line'owe narzędzie, którego zadaniem jest tworzenie projektów z tak zwanych cookiecutters, czyli gotowych szablonów, za pomocą, których Cookiecutter umie stworzyć szablon paczki.

W tym celu użyty jest system templatowanie Jinja2, która potrafi inteligentnie zastąpić imiona i strukturę folderów oraz plików i ich zawartości. Co bardzo dobrze pokazuje poniżej zamieszczony przykład:



Rys. 5.1: Przykład działania Cookiecutter'a [28]

5.3. Tworzenie paczki

Używając gotowego szablonu Cookiecutter'a 5.2 [44] tworzymy szkielet naszej paczki.

Listing 5.1: Użycie Cookiecutter'a

```
$ cookiecutter gh:giswqs/pypackage
full_name [Qiusheng Wu]: Jan Bronicki
email [admin@example.com]: janbronicki@gmail.com
github_username [giswqs]: John15321
project_name [Python Boilerplate]: Simple Game AI
project_slug [simple_game_ai]: sgai
project_short_description: Simple Game AI package allows to easily define an
interface for a game and train a simple neural net to play it
pypi_username [John15321]: John15321
version [0.0.1]: 0.0.1
use_pytest [n]:
add_pyup_badge [n]:
create_author_file [n]:
Select command_line_interface:
1 - No command-line interface
2 - Click
3 - Argparse
Choose from 1, 2, 3 [1]: 1
Select open_source_license:
1 - MIT license
2 - BSD license
3 - ISC license
4 - Apache Software License 2.0
5 - GNU General Public License v3
6 - Not open source
Choose from 1, 2, 3, 4, 5, 6 [1]: 1
Select github_default_branch:
1 - main
2 - master
Choose from 1, 2 [1]: 2
```

Wybrany szablon przygotowuje nam wiele narzędzi takie jak na przykład `mkdcos`, narzędzie służące do generowania dokumetnacji w języku Markdown. Dodatkowo tworzy on automatycznie configurację CI/CD Pipeline.

Za pomocą komendy `tree` możemy zobczyć jaką strukturę paczki wygenerował, dla nas Cookiecutter 5.2.

Listing 5.2: Wygenerowana struktura paczki za pomocą Cookiecutter'a

```
$ tree
.
|-- docs
|   |-- contributing.md
|   |-- faq.md
|   |-- index.md
|   |-- installation.md
|   |-- overrides
|   |   `-- main.html
|   |-- sgai.md
|   `-- usage.md
|-- LICENSE
|-- MANIFEST.in
|-- mkdocs.yml
|-- README.md
|-- requirements_dev.txt
|-- requirements.txt
|-- setup.cfg
|-- setup.py
|-- sgai
|   |-- __init__.py
|   `-- sgai.py
`-- tests
    |-- __init__.py
    `-- test_sgai.py
```

Widzimy, że Cookiecutter stworzył (zaczynając alfabetycznie), folder `docs` w, którym mieści się dokumentacja napisana w języku Markdown, plik `LICENSE` opisujący licencję MIT [9] na, której wydana została paczka, plik `MANIFEST.in`, który opisuje dodatkowe pliki zawarte w paczce, które nie są kodem źródłowym takie jak `README.md` i `requirements.txt`, plik konfiguracyjny `mkdocs.yml` odpowiadający za konfiguracje dokumentacji Markdown, plik `README.md` opisujący podstawowe informacje apropos paczki, `requirements.txt`, który wskazuje pip'owi wymagane zależności wymagane, dla poprawnego działania paczki, `requirements_dev.txt`, czyli wymagane zależności do pracy nad paczką, takie jak formater kodu `black`, lub `pytest` służący do automatyzowania testów, `setup.cfg`, który funkcjonuje jako plik konfiguracyjny, dla `setup.py`. Następnie widzimy folder `sgai`, który jest miejscem na kod źródłowy.

Tak prezentuje się struktura projektu po dodaniu odpowiednich plików kodu źródłowego, gdzie najważniejsze to 5.3:

- `agent.py` - tworzy i reguluje parametry sieci neuronowej,
- `config.py` - posiada konfiguracje parametrów procesu nauki sieci neuronowej takie jak `LEARNING_RATE`,
- `data_helper.py` - funkcje odpowiedzialne za rysowanie wykresów wizualizujących postępy nauki sieci neuronowej,
- `model.py` - posiada model sieci neuronowej i jej mechanizmy nauki,
- `trainer.py` - definiuje interfejs klasy abstrakcyjnej za pomocą, której użytkownik paczki łączy grę z treningiem sieci neuronowej.

Listing 5.3: Struktura gotowej paczki

```
$ tree
.
|-- AUTHORS.rst
|-- docs
|   |-- authors.rst
|   |-- contributing.md
|   |-- faq.md
|   |-- index.md
|   |-- installation.md
|   |-- overrides
|   |   `-- main.html
|   |-- sgai.md
|   `-- usage.md
|-- LICENSE
|-- MANIFEST.in
|-- mkdocs.yml
|-- README.md
|-- requirements_dev.txt
|-- requirements.txt
|-- setup.cfg
|-- setup.py
|-- sgai
|   |-- agent
|   |   |-- agent.py
|   |   |-- config.py
|   |   |-- data_helper.py
|   |   |-- __init__.py
|   |   |-- model.py
|   |   `-- trainer.py
|   |-- __init__.py
|   `-- sgai.py
`-- tests
    |-- __init__.py
    `-- test_sgai.py
```

5.4. Publikowanie paczki

Następnie za pomocą skryptu `setup.py` tworzymy źródłową dystrybucje paczki, uzyskując plik z rozszerzeniem `.tar.gz` 5.4.

Listing 5.4: Generacja paczki

```
$ python setup.py sdist
...
Writing sgai-0.0.3/setup.cfg
creating dist
Creating tar archive
```

Owy plik znajduje się w nowo powstałym folderze `dist` w root'cie projektu 5.5.

Listing 5.5: Zbudowana source distribution paczki

```
$ tree
.
...
|-- dist
|   `-- sgai-0.0.3.tar.gz
...
```

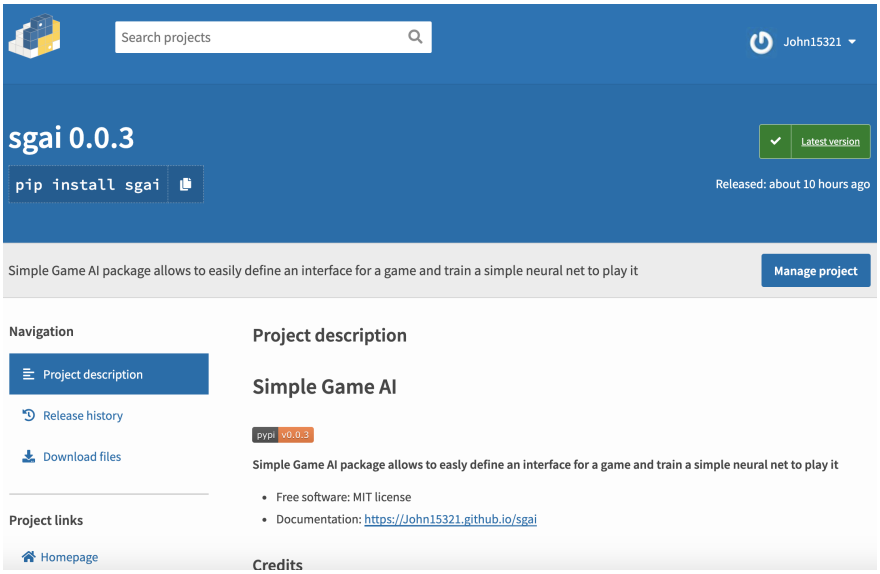
Następnie za pomocą narzędzia `twine` uploadujemy wygenerowany plik do indeksu PyPI 5.1 5.6.

Listing 5.6: Upload zbudowanej paczki na PyPI

```
$ twine upload ./dist/sgai-0.0.3.tar.gz
Uploading distributions to https://upload.pypi.org/legacy/
Enter your username: John15321
Enter your password:
Uploading sgai-0.0.3.tar.gz

View at:
https://pypi.org/project/sgai/0.0.3/
```

Otwierając stronę pod adresem `https://pypi.org/project/sgai` widoczna i gotowa to instalacji jest paczka `sgai` 5.2.



Rys. 5.2: Paczka sgai widoczna na stronie PyPI

Aby sprawdzić czy paczka została poprawnie dodana do globalnego indeksu PyPI w nowym środowisku możemy spróbować zainstalować paczkę `sgai` w następujący sposób 5.7.

Listing 5.7: Instalacja paczki używając narzędzia pip

```
$ pip install sgai

Collecting sgai
  Downloading sgai-0.0.3.tar.gz (7.4 kB)
  Preparing metadata (setup.py) ... done
  ...
```

Po zakończonej sukcesem instalacji, możemy sprawdzić czy pip widzi naszą paczke i jej wersje wykonując poniższą komende 5.8.

Listing 5.8: Wylistowanie paczki za pomocą narzędzia pip

```
$ pip list

Package            Version
------------------ -------
...
sgai               0.0.3
...
```

Korzystając z interaktywnego interpretera Python'a, IPython, możemy spróbować zaimportować zainstalowaną paczkę i wyświetlić jej metadane 5.9.

Listing 5.9: Importowanie zainstalowanej paczki

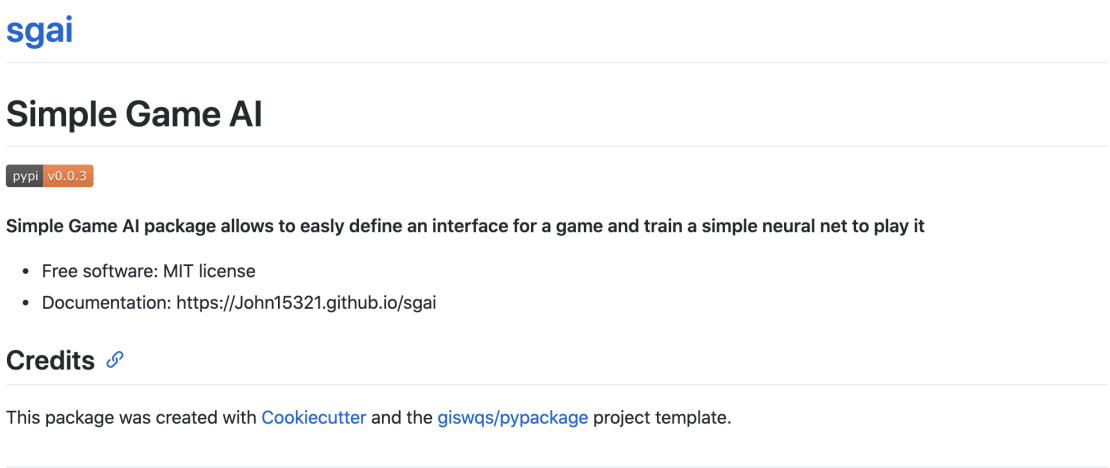
```
$ ipython

In [1]: import sgai


In [2]: sgai?
Type:        module
String form: <module 'sgai' from '/Users/jbron/sgai/sgai/__init__.py'>
File:        ~/sgai/sgai/__init__.py
Docstring:   Top-level package for Simple Game AI.
```

Wszystko się zgadza co oznacza, że paczka została poprawnie udostępniona i zainstalowana.

Dodatkowo możemy zobaczyć sporządzoną dokumentację na temat paczki na stronie `https://john15321.github.io/sgai/`, która jest hostowana dzięki serwisowy GitHub na, którym znajduje się kod źródłowy paczki 5.3.



Rys. 5.3: Dokumentacja paczki sgai

Rozdział 6

Użycie

6.1. Gra Snake

Snake [24] jest popularnym konceptem prostej gry komputerowej, gdzie gracz ma za zadanie manewrować poruszającą się linią w taki sposób, aby nie kolidowała ze sobą lub, dodatkowo ściankami planszy, w między czasie zbierając punkty losowo porozmieszczane po mapie. Linia ma za zadanie reprezentować ciało węża, który pełza, a punkty reprezentowane są w postaci owocu, lub prostej grupki pikseli. Odmiana użyta w celu testowania sieci neuronowej została na pisana w języku Python 3.1, przy użyciu paczki PyGame 3.5, która posłuży do reprezentacji graficznej postępowania sztucznej inteligencji, oraz samej rozgrywki. Wąż został przestawiony w postaci linii składającej się z pojedynczych zielonych kwadratów. Wąż zaczyna z trzema zielonymi kwadratami, każdy dodatkowo zdobyty punkt podłuża węża o jeden kwadrat. Punkty możliwe do zdobycia są reprezentowany w postaci czerwonych kwadratów 6.1.

Rys. 6.1: Gra Snake

6.2. Implementacja gry

Na początku zaimportowane zostają moduły potrzebne 6.1 do uruchomienia gry i do jej wewnętrznych procesów.

Listing 6.1: Importowanie modułów do gry Snake

```
import random
from enum import Enum
from typing import Tuple

import numpy as np
import pygame
```

Stworzenie pomocniczych klas 6.2 służących jako enumeratory na kolory i kierunki.

Listing 6.2: Pomocnicze enumeratory gry Snake

```
class GameColors(Enum):
    WHITE: Tuple = (255, 255, 255)
    RED: Tuple = (150, 0, 0)
    DARK_GREEN: Tuple = (0, 60, 10)
    LIGHT_GREEN: Tuple = (50, 160, 80)
    BLACK: Tuple = (0, 0, 0)


class GameDirection(Enum):
    RIGHT = 1
    LEFT = 2
    UP = 3
    DOWN = 4
```

Klasa tworząca abstrakcję punktu na mapie powstałego z pikseli 6.3.

Listing 6.3: Klasa odpowiadająca za punkty na mapie

```
class GamePointOnTheMap:
    def __init__(self, x: int, y: int) -> None:
        self._x = None
        self._y = None

        self.x = x
        self.y = y

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        return False

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y

    @x.setter
    def x(self, x):
        self._x = x

    @y.setter
    def y(self, y):
        self._y = y
```

Definicja głównej klasy odpowiedzialnej za mechanikę gry. W jej konstruktorze zainicjalizowane zostaje okno graficzne służące wizualizacji gry 6.4.

Listing 6.4: Klasa implementują mechanikę gry Snake

```
class ExampleSnakeGame:
    def __init__(self, w: int = 640, h: int = 480):
        self.w = w
        self.h = h
        self.display = pygame.display.set_mode((self.w, self.h))
        pygame.display.set_caption("Snake")
        self.clock = pygame.time.Clock()
        self.reset_game_state()
```

Funkcja odpowiadająca za resetowanie gry 6.5.

Listing 6.5: Funkcja resetująca rozgrywke

```
def reset_game_state(self):
    # Initialize game state
    self.direction = GameDirection.RIGHT

    self.head = GamePointOnTheMap(self.w / 2, self.h / 2)
    self.snake = [
        self.head,
        GamePointOnTheMap(self.head.x - BLOCK_SIZE, self.head.y),
        GamePointOnTheMap(self.head.x - (2 * BLOCK_SIZE), self.head.y),
    ]

    self.score = 0
    self.food = None
    self._place_food()
    self.frame_iteration_number = 0
```

Prywatna funkcja odpowiadająca za losowanie miejsca pojawienia się jedzenia, dla węża na mapie 6.6.

Listing 6.6: Funkcja umieszająca jedzenie na mapie

```
def _place_food(self):
    x = random.randint(0, (self.w - BLOCK_SIZE) // BLOCK_SIZE) * BLOCK_SIZE
    y = random.randint(0, (self.h - BLOCK_SIZE) // BLOCK_SIZE) * BLOCK_SIZE
    self.food = GamePointOnTheMap(x, y)
    if self.food in self.snake:
        self._place_food()
```

Funkcja `play_step()` jest odpowiedzialna za każdą klatkę rozgrywki 6.7, na początku sprawdza czy użytkownik się chce wyjść z rozgrywki, następnie wykonuje ruchy węża jakie miały nastąpić. Na podstawie wykonanego ruchu ocenia sytuację rozgrywki, i decyduje o tym jaka powinna być nagroda za dany ruch. Gdzie zdobycie jedzenia to dodatkowe 10 punktów, nie zdobycie jedzenia to 0 punktów, a śmierć ujemne 10 punktów. Następnie zakładając, że wąż żyje, jedzenie jest umieszczane w innym miejscu, lub wąż wykonuje ruch. Po czym następuje aktualizacja interfejsu o kolejną klatkę i zwrócenie informacji o danej klatce takich jak nagroda i stan gry.

Listing 6.7: Funkcja implementacja mechanikę rozgrywki klatka po klatce

```
def play_step(self, action) -> Tuple[int, bool, int]:
    self.frame_iteration_number += 1
    # 1. collect user input
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
    # 2. move
    self._move(action)  # update the head
    self.snake.insert(0, self.head)
    # 3. Update the game state
    # Eat food: +10
    # Game Over: -10
    # Else: 0
    reward = 0
    game_over = False
    if self.is_collision() or self.frame_iteration_number>100*len(self.snake):
        game_over = True
        reward = -10
        return reward, game_over, self.score
    # 4. place new food or just move
    if self.head == self.food:
        self.score += 1
        reward = 10
        self._place_food()
    else:
        self.snake.pop()
    # 5. update ui and clock
    self._update_ui()
    self.clock.tick(SPEED)
    # 6. return game over and score
    return reward, game_over, self.score
```

Pomocnicza funkcja, której zadaniem jest wykrycie kolizji obiektów 6.8.

Listing 6.8: Funkcja sprawdzająca kolizje

```
def is_collision(self, point=None):
    if point is None:
        point = self.head
    # Check if it hits the boundary
    if (
        point.x > self.w - BLOCK_SIZE
        or point.x < 0
        or point.y > self.h - BLOCK_SIZE
        or point.y < 0
    ):
        return True
    # Check if it hits itself
    if point in self.snake[1:]:
        return True
    return False
```

Pomocnicza prywatna funkcja odpowiedzialna za aktualizację interfejsu graficznego 6.9.

Listing 6.9: Funkcja uaktualniająca interfejs

```
def _update_ui(self):
    self.display.fill(GameColors.BLACK.value)
    for pt in self.snake:
        pygame.draw.rect(
            self.display,
            GameColors.DARK_GREEN.value,
            pygame.Rect(pt.x, pt.y, BLOCK_SIZE, BLOCK_SIZE),
        )
        pygame.draw.rect(
            self.display,
            GameColors.LIGHT_GREEN.value,
            pygame.Rect(pt.x + 4, pt.y + 4, 12, 12),
        )
    pygame.draw.rect(
        self.display,
        GameColors.RED.value,
        pygame.Rect(self.food.x, self.food.y, BLOCK_SIZE, BLOCK_SIZE),
    )
    text = font.render(f"Score: {self.score}", True, GameColors.WHITE.value)
    self.display.blit(text, [0, 0])
    pygame.display.flip()
```

Pomocnicza prywatna funkcja odpowiadająca za obliczanie następnej pozycji węża 6.10.

Listing 6.10: Funkcja poruszającą wężem

```
def _move(self, action):
    # [straight, right, left]
    clock_wise = [
        GameDirection.RIGHT,
        GameDirection.DOWN,
        GameDirection.LEFT,
        GameDirection.UP,
    ]
    move_index = clock_wise.index(self.direction)
    if np.array_equal(action, [1, 0, 0]):
        new_direction = clock_wise[move_index]  # no change
    elif np.array_equal(action, [0, 1, 0]):
        next_idx = (move_index + 1) % 4
        new_direction = clock_wise[next_idx]
    else:  # [0, 0, 1]
        next_idx = (move_index - 1) % 4
        new_direction = clock_wise[next_idx]
    self.direction = new_direction
    x = self.head.x
    y = self.head.y
    if self.direction == GameDirection.RIGHT:
        x += BLOCK_SIZE
    elif self.direction == GameDirection.LEFT:
        x -= BLOCK_SIZE
    elif self.direction == GameDirection.DOWN:
        y += BLOCK_SIZE
    elif self.direction == GameDirection.UP:
        y -= BLOCK_SIZE
    self.head = GamePointOnTheMap(x=x, y=y)
```

6.3. Przykład użycia

Większość mechaniki gry została zaimplementowana w klasie `ExampleSnakeGame` 6.4, która odpowiada za mechanikę gry oraz przechowywanie jej stanu. Znajdują się w niej między innymi mechaniki odpowiedzialne za wykrywanie zderzeń, poruszanie się węża, sterowanie wężem, czy losowanie miejsca pojawienia się następnych punktów. Dodatkowo zaimplementowane zostały klasy enumeratorowe, takie jak `GameDirection` 6.2 i `GamePointOnTheMap` 6.3 w celu zwiększenia czytelności kodu 6.11.

Listing 6.11: Importowanie gry

```
import numpy as np
import torch
from sgai.agent.trainer import Trainer

from snake import GameDirection, GamePointOnTheMap, ExampleSnakeGame
```

Następnie definiujemy potrzebny interfejs `Trainer` 4.10, aby paczka `sgai` umiała się komunikować z grą. Zaczynami od definicji klasy i jej konstruktora. W tym przykładzie wielkość warstwy ukrytej pozostawiamy nie zmienioną ponieważ podstawowa w pełni wystarczy do poziomu skomplikowania przygotowanej gry 6.12.

Listing 6.12: Definicja klasy `MyTrainer`

```
class MyTrainer(Trainer):
    def __init__(self, game, input_size: int, output_size: int, hidden_size: int):
        super().__init__(game, input_size=input_size, output_size=output_size, hidde
```

Następnie definiujemy dwie funkcje `get_state()` 6.13, oraz `perform_action()` 6.15. Są one interfejsem pomiędzy paczką `sgai`, a kodem gry, dlatego to użytkownik paczki musi je zaimplementować, ponieważ paczka nie ma możliwości posiadania wiedzy na temat własności działania kodu gry na, której będzie szkolić sieć neuronową. Na samym początku definiujemy stan gry, pobierając konkretne elementy rozgrywki takie jak pozycja węża, pozycja nagrody itp..

Listing 6.13: Definicja funkcji `get_state()`

```
    def get_state(self) -> np.ndarray:
        head = self.game.snake[0]
        point_left = GamePointOnTheMap(head.x - 20, head.y)
        point_right = GamePointOnTheMap(head.x + 20, head.y)
        point_up = GamePointOnTheMap(head.x, head.y - 20)
        point_down = GamePointOnTheMap(head.x, head.y + 20)

        left_direction = self.game.direction == GameDirection.LEFT
        right_direction = self.game.direction == GameDirection.RIGHT
        up_direction = self.game.direction == GameDirection.UP
        down_direction = self.game.direction == GameDirection.DOWN
```

W tym przykładzie zebranych będzie łącznie 11 parametrów 6.14 opisujących stan gry, które następnie zostaną przekazane sieci neuronowej jako wejście. Gdzie każda z wartości może być jedynie 0, lub 1, 0 oznaczające `False`, a 1 `True`.

- Czy jakieś niebezpieczeństwo znajduje się naprzeciwko?
- Czy niebezpieczeństwo znajduje się po prawej?
- Czy niebezpieczeństwo znajduje się po lewej?
- Czy wąż porusza się w lewo?
- Czy wąż porusza się w prawo?
- Czy wąż porusza się w górę?
- Czy wąż porusza się w dół?
- Czy jedzenie znajduje się po lewej?
- Czy jedzenie znajduje się po prawej?
- Czy jedzenie znajduje się nad wężem?
- Czy jedzenie znajduje się pod wężem?

Listing 6.14: Zwracany stan

```
state = [
    # Danger is straight ahead
    (right_direction and self.game.is_collision(point_right))
    or (left_direction and self.game.is_collision(point_left))
    or (up_direction and self.game.is_collision(point_up))
    or (down_direction and self.game.is_collision(point_down)),
    # Danger is on the right
    (up_direction and self.game.is_collision(point_right))
    or (down_direction and self.game.is_collision(point_left))
    or (left_direction and self.game.is_collision(point_up))
    or (right_direction and self.game.is_collision(point_down)),
    # Danger is on the left
    (down_direction and self.game.is_collision(point_right))
    or (up_direction and self.game.is_collision(point_left))
    or (right_direction and self.game.is_collision(point_up))
    or (left_direction and self.game.is_collision(point_down)),
    # Current move direction
    left_direction,
    right_direction,
    up_direction,
    down_direction,
    # Food location
    self.game.food.x < self.game.head.x,  # Food is on the left
    self.game.food.x > self.game.head.x,  # Food is on the right
    self.game.food.y < self.game.head.y,  # Food is up
    self.game.food.y > self.game.head.y,  # Food is down
]

return np.array(state, dtype=int)
```

Ponieważ wewnętrzna funkcja `play_step()` gry sama zwraca nagrodę, stan oraz wynik gry (które są wynikiem podjętej przez nas akcji), zaimplementowanie funkcji `perform_action()` jest bardzo proste.

Listing 6.15: Implementacja funkcji `perform_action()`

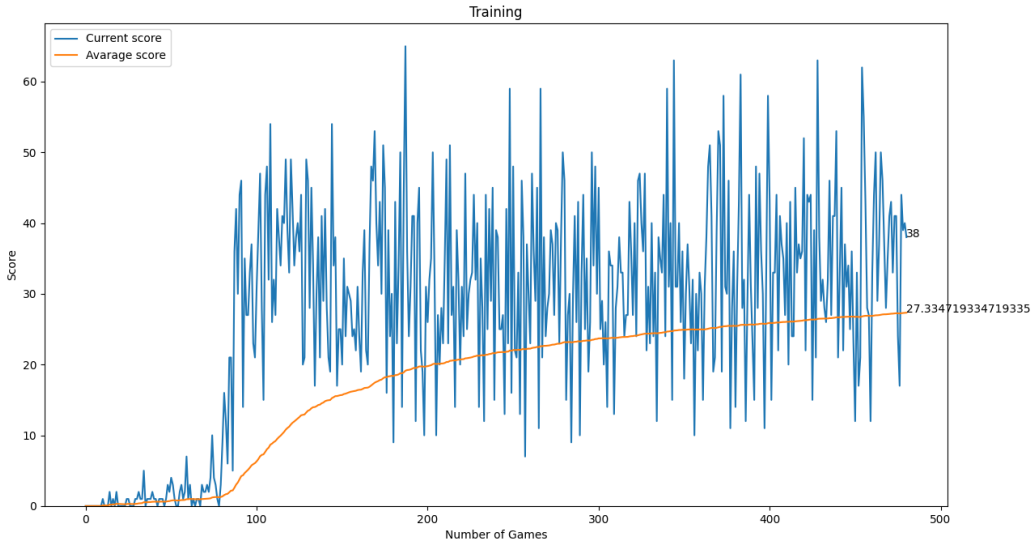
```
def perform_action(self, final_move) -> Tuple[int, bool, int]:
    reward, game_over, score = self.game.play_step(final_move)
    return reward, game_over, score
```

Tworzymy obiekt klasy `MyTrainer` 6.16, definiujemy jego parametry i uruchamiamy trening sieci neuronowej.

Listing 6.16: Implementacja klasy Trainer

```
if __name__ == "__main__":
    mt = MyTrainer(
        game=ExampleSnakeGame(), input_size=11, output_size=3, hidden_size=256
    )
    mt.train(model_file="model.pth")
```

Jak widać na poniższym wykresie 6.2 w okolicach setnej rozgrywki, sieć neuronowa znacznie się poprawia. Następnie przez kolejne kilkaset gier widoczny jest jedynie marginalny postęp, przez to można stwierdzić że nastąpiło nasycenie spowodowane prostą budową sieci.



Rys. 6.2: Wykres nauki sieci neuronowej

Rozdział 7

Podsumowanie

Celem pracy było zaprojektowanie oraz stworzenie sieci neuronowej zdolnej do samodzielnej nauki gry w dowolną prostą grę zręcznościową. Cel został w całości zrealizowany. Model sieci neuronowej został oparty o uczenie przez wzmacnianie w połączeniu z Q-learning'iem. Dzięki użyciu abstrakcyjnych klas i funkcji języka Python, udało się stworzyć zgeneralizowany interfejs służący do trenowania sieci neuronowej. Dzięki czemu użytkownik poprawnie definiując odpowiednie funkcje wejścia i wyjścia sieci neuronowej w połączeniu ze stanem, oraz oddziaływaniem na grę, jest w stanie trenować sieć neuronową na dowolnej grze.

Użyto narzędzia Cookiecutter do stworzenia szablonu paczki Python'owej. Implementację sieci neuronowej przygotowano w sposób umożliwiający umieszczenie jej wewnątrz paczki języka Python. Z której następnie pomyślnie udało się stworzyć dystrybucję źródłową paczki. Następnie zarejestrowano paczkę na oficjalnym indeksie paczek języka Python, PyPI, i pomyślnie ją tam umieszczono.

Testy zostały wykonane przy pomocy stworzonej na potrzeby projektu gry typu Snake. Pomyślnie zaimplementowano interfejs pomiędzy grą a siecią neuronową z paczki. Podczas testów sieć uzyskała zadowalające wyniki, jak na swoją prostą budowę.

Projekt można nadal rozwijać poprzez ulepszenie generacji struktury sieci neuronowej tak, aby ta była w stanie się dostosować do bardziej skomplikowanych gier. Dodatkową możliwością byłoby ułatwienie implementacji interfejsu pomiędzy grą a siecią tak, aby była ona bardziej przyjazna, dla użytkownika paczki.

Literatura

[1] Pep 241 – metadata for python software packages. https://www.python.org/dev/peps/pep-0241/, 2001.

[2] Artificial neural network. https://en.wikipedia.org/wiki/Artificial_neural_network, 2021.

[3] Backpropagation. https://en.wikipedia.org/wiki/Backpropagation, 2021.

[4] Bellman equation. https://en.wikipedia.org/wiki/Bellman_equation, 2021.

[5] Błąd średniokwadratowy. https://pl.wikipedia.org/wiki/B%C5%82%C4%85d_%C5%9Bredniokwadratowy, 2021.

[6] Colored neural network. https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg, 2021.

[7] Cookiecutter: Better project templates. https://cookiecutter.readthedocs.io/, 2021.

[8] Matplotlib. https://en.wikipedia.org/wiki/Matplotlib, 2021.

[9] Mit license. https://en.wikipedia.org/wiki/MIT_License, 2021.

[10] Neural network. https://en.wikipedia.org/wiki/Neural_network, 2021.

[11] Neuroplastyczność. https://pl.wikipedia.org/wiki/Neuroplastyczno%C5%9B%C4%87, 2021.

[12] Numpy. https://en.wikipedia.org/wiki/NumPy, 2021.

[13] Numpy manual. https://numpy.org/doc/stable/, 2021.

[14] pip (package manager). https://en.wikipedia.org/wiki/Pip_(package_manager), 2021.

[15] Pygame. https://en.wikipedia.org/wiki/Pygame, 2021.

[16] Pygame documentation. https://www.pygame.org/wiki/GettingStarted, 2021.

[17] Python package index. https://en.wikipedia.org/wiki/Python_Package_Index, 2021.

[18] Pytorch. https://pytorch.org/docs/stable/index.html, 2021.

[19] Pytorch documentation: Adam. https://pytorch.org/docs/stable/generated/torch.optim.Adam.html, 2021.

[20] Q-learning. https://en.wikipedia.org/wiki/Q-learning, 2021.

[21] Readmme.md. https://github.com/cookiecutter/cookiecutter, 2021.

[22] Reinforcement learnin. https://en.wikipedia.org/wiki/Reinforcement_learning, 2021.

[23] Równanie bellman'a. https://cdn-media-1.freecodecamp.org/images/1*jmcVWHHbzCxDc-irBy9JTw.png, 2021.

[24] Snake (video game genre). https://en.wikipedia.org/wiki/Snake_(video_game_genre), 2021.

[25] Users guide. https://matplotlib.org/stable/users/index, 2021.

[26] E. Bressert. *SciPy and NumPy*. O'Reilly Media, Inc., 2012.

[27] J. Brownlee. Gentle introduction to the adam optimization algorithm for deep learning. 2017.

[28] H. Canto. Project templates and cookiecutter. https://medium.com/worldsensing-techblog/project-templates-and-cookiecutter-6d8f99a06374, 2018.

[29] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. O'Reilly Media, Inc., 2019.

[30] N. Habib. *Hands-On Q-Learning with Python*. Packt Publishing, 2019.

[31] M. Lapan. *Deep Reinforcement Learning Hands-On - Second Edition*. Packt Publishing, 2020.

[32] B. Lubanovic. *Introducing Python, 2nd Edition*. O'Reilly Media, Inc., 2019.

[33] E. S. Luca Pietro Giovanni Antiga, Thomas Viehmann. *Deep Learning with PyTorch*. Manning Publications, 2020.

[34] W. McKinney. *Python for Data Analysis, 3rd Edition*. O'Reilly Media, Inc., 2021.

[35] T. Z. Michal Jaworski. *Expert Python Programming - Fourth Edition*. Packt Publishing, 2021.

[36] B. normalization. Batch normalization. https://en.wikipedia.org/wiki/Batch_normalization, 2021.

[37] J. Papa. *PyTorch Pocket Reference*. O'Reilly Media, Inc., 2021.

[38] I. Pointer. *Programming PyTorch for Deep Learning*. O'Reilly Media, Inc., 2019.

[39] S. R. Poladi. *Matplotlib 3.0 Cookbook*. Packt Publishing, 2018.

[40] S. Weidman. *Deep Learning from Scratch*. O'Reilly Media, Inc., 2019.

[41] Wikipedia. Python (programming language). `https://en.wikipedia.org/wiki/Python_(programming_language)`.

[42] M. Wilkes. *Advanced Python Development: Using Powerful Language Features in Real-World Applications*. Apress, 2020.

[43] P. Winder. *Reinforcement Learning*. O'Reilly Media, Inc., 2020.

[44] Q. Wu. Cookiecutter pypackage. `https://github.com/giswqs/geodemo`, 2021.

[45] A. Zamczała. Python ma już 30 lat! od czego wszystko się zaczęło? 2021.

Spis rysunków

Spis tabel