AKADEMIA GÓRNICZO-HUTNICZA
im. S. Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

“Symulacja autonomicznego ruchu samochodu po torze”

Kierunek:
Informatyka:
Przedmiot:
Studio projektowe
Prowadzący:
prof. dr. hab. Andrzej Bielecki
Autorzy:
Krzysztof Pala
Szymon Czaplak
Alicja Brajner
Data utworzenia:
Semestr letni 2019/2020

1. WSTĘP

1.1 Opis projektu

Celem projektu była próba zrobienia inteligentnego modelu sterującego autem w symulacji 2D. W tym celu zostało stworzone środowisko, które prezentowało poruszanie się modelu samochodu po zaprojektowanym torze oraz zbierało dane dotyczące przejazdu i kolizji. Dodatkowo projekt miał nas nauczyć pracy z inteligentnymi modelami, narzędziami do ich kreacji oraz różnych częstych problemów w dziedzinie machine learningu.

1.2 Zastosowane technologie

Projekt został zrealizowany w języku **Python** w środowisku **PyCharm**. Zostały zastosowane między innymi biblioteki:

- **pygame** - biblioteka do tworzenia gier. Posłużyła celu wizualizacji poruszania się auta po torze.
- **keras, tensorflow -** biblioteka zawierająca narzędzia głębokich sieci neuronowych i uczenia maszynowego.
- **pandas -** biblioteka służąca do analizy i manipulacji danych.
- **numpy, scipy -** biblioteki zawierające obsługę dużej ilości funkcji matematycznych

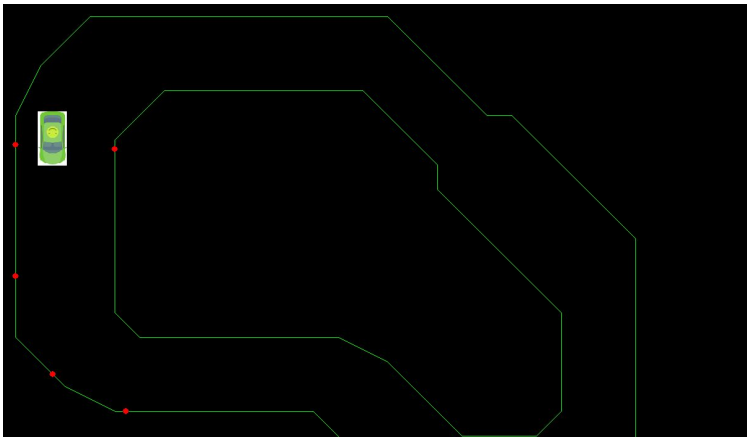
1.3 Podział pracy

Przygotowanie środowiska symulacji: Alicja Brajner
Przygotowanie modelu zbierania danych, zebranie danych: Krzysztof Pala
Prosty algorytm nauki w sieci neuronowej: Krzysztof Pala, Szymon Czaplak
Algorytm wyboru architektury sieci za pomocą alg. gen: Szymon Czaplak
Algorytm stworzenia modelu z zastosowaniem alg. gen. do modyfikacji wag: Szymon Czaplak, Krzysztof Pala, Alicja Brajner

2. REALIZACJA PROJEKTU

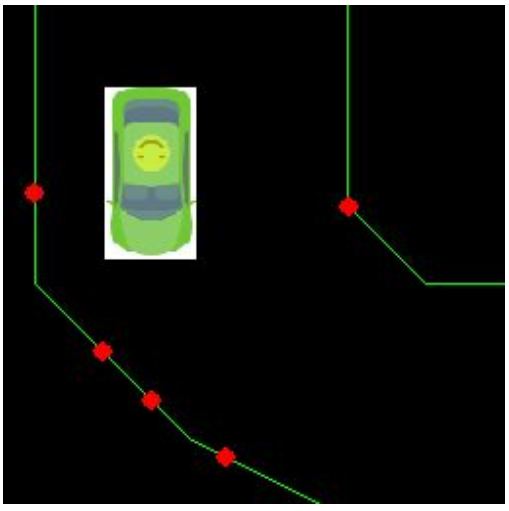
2.1 Stworzenie środowiska symulacji

W pierwszej kolejności stworzone zostało środowisko symulacji oraz jego wizualizacja za pomocą biblioteki **pygame**. Tor został zaprojektowany za pomocą rysowania prostych linii na wybranej przestrzeni w kwadratowym oknie. Program umożliwiał poruszanie się autem przez użytkownika za pomocą wejścia klawiatury. Reagował na zderzenia auta z ścianą toru.



2.2 Zebranie danych do utworzenia modelu

Jako dane do utworzenia modelu postanowiliśmy zebrać odległości auta do najbliższej ściany toru. W tym celu został stworzony mechanizm “sensorów”, który zbierał dane: odległości do ściany na wprost auta, 80 i 15 stopni z lewej i prawej strony auta.



Dane z sensorów wraz z podjętą przez gracza decyzją zostały zebrane do plików csv w następującej formie:

```
394.0,101.54219270825305,60.92566290160494,470.0156619730879,231.819
25373014212,straight
```

Dane oznaczały po kolei: pomiary sensorów, oraz decyzję ws. poruszania się samochodu.

2.3 Stworzenie modelu za pomocą sieci neuronowych

Następnym etapem projektu była próba stworzenia modelu za pomocą sieci neuronowych. Model miał za zadanie w każdej klatce działania gry przewidzieć klasę decyzji: skręt w lewo/skręt w prawo, lub jazdę prosto. Po próbach trenowania różnych architektur sieci najlepsze rezultaty otrzymaliśmy stosując sieć z jedna warstwą z 4000 neuronami. Algorytm dobrze sobie radził z symulacją jazdy w jedną stronę toru - zgodną z kierunkiem jazdy w zebranych danych - ale zarówno przy zmianie kierunku poruszania się, jak i zmianie toru radził sobie fatalnie. Poniżej fragment kodu z wykorzystaniem architektury sieci z biblioteki **keras**.

```
network = keras.Sequential()
network.add(Dense(4000, activation='relu', input_shape=(5,)))
network.add(Dense(3, activation='softmax'))
optimizer = optimizers.Adam(learning_rate=0.001, beta_1=0.9,
beta_2=0.999, amsgrad=False)
network.compile(optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy'])
```

Ta próba pozwoliła nam wysnuć wnioski, że prawdopodobnie mamy do czynienia z overfittingiem. Tak duża sieć neuronowa tylko zapamiętała i odtwarzała działania użytkownika z danych treningowych, przez co była bezużyteczna w innych, nowych warunkach. Postanowiliśmy spróbować następujących rozwiązań:

- zdywersyfikowanie danych do stworzenia modelu - użycie innych torów, zmiana kierunku.
- znalezienie optymalnej architektury sieci za pomocą algorytmu genetycznego

2.4 Zastosowanie algorytmu genetycznego do poszukiwania najlepszej architektury sieci

Jednym z napotkanych wcześniej problemów był zły dobór architektury sieci neuronowej. Postanowiliśmy wykorzystać algorytm genetyczny do wybrania najbardziej optymalnej z nich. W tym celu stworzyliśmy obiekt, który umożliwiał dynamicznie inicjalizowanie obiektu o dowolnej strukturze z zachowaniem odpowiednich wymiarów danych. Postanowiliśmy wypróbować architekturę sieci LSTM, ze względu na to, że są skuteczne w nauce sekwencyjnych danych jakimi takie dane z autka są. Wprowadziliśmy też polynomial features, które umożliwiają tworzenie wielomianów z tych 5 liczb, które mamy od 1 do 3 stopnia max.

```
self.epochs = 25  # liczba epok
self.lstm_layers_number_range = (1, 3)  # ilość warstw LSTM
self.lstm_neurons_range = (16, 128) # ilość neuronów w warstwie LSTM
self.dense_layers_number_range = (1, 4) # ilość warstw Dense (fully
connected)
self.dense_neurons_range = (10, 400)# tutaj ile neuronów może być w
warstwie dense
self.polynomial_number_range = (1, 3) # zakres stopnia wielomianu w
preprocessingu
```

Zdecydowaliśmy się na następujące właściwości algorytmu genetycznego:

- ewaluacja jednostki: ilość “przeżytych” klatek do momentu zderzenia przy próbie zastosowanie modelu w symulacji.
- prawdopodobieństwo mutacji: 10%
- mutacja: zmiana jednego losowego parametru w ustalonych zakresach
- wskaźnik selekcji: 20% najlepszych osobników
- krzyżowanie: losowy wybór jednego z parametrów sieci ***['lstm_layers_number', 'dense_layers_number', 'polynomial_exponent'],*** a następnie zmiana na wartość jednego z rodziców, lub średniej wartości obu rodziców.
- losowy dobór par

Jak można było się spodziewać algorytm na komputerze personalnym przy ograniczonej mocy CPU działał około 24h. Najlepszy stworzony model nie dał jednak zadowalających rezultatów. Po trzech nieudanych próbach ze względu na ograniczoną moc obliczeniową i czas użytkowania komputerów osobistych postanowiliśmy nie rozwijać dalej tego rozwiązania.

2.5 Zastosowanie algorytmu genetycznego w małej sieci neuronowej do modyfikacji wag w sieci

W następnym podejściu postanowiliśmy także wykorzystać algorytm genetyczny, jednak już w samej sieci neuronowej - do modyfikacji wag bezpośrednio w sieci. W tym celu zrezygnowaliśmy z używanych wcześniej frameworkowych sieci neuronowych i częściowo zaprojektowaliśmy własną sieć.

Stworzyliśmy sieć zawierającą już nie 4000, lecz 4 neurony.

```
self.nn_architecture = [
  {"input_dim": 5, "output_dim": 4, "activation": "relu"},
  {"input_dim": 4, "output_dim": 3, "activation": "sigmoid"},
]
```

Czyli jedna warstwa 4 neuronów, przyjmująca wartości z sensorów auta oraz warstwa outputowa, która odpowiada za wskazanie odpowiedniej czynności.

Do modyfikacji wag zastosowaliśmy algorytm genetyczny o następujących właściwościach:

- ewaluacja jednostki: ilość “przeżytych” klatek do momentu zderzenia przy próbie zastosowanie modelu w symulacji.
- prawdopodobieństwo mutacji: 10%
- mutacja: losowe ustawienie wag w całej sieci
- wskaźnik selekcji: 20% najlepszych osobników
- krzyżowanie: średnia wag mężczyzny i kobiety
- losowy dobór par

Utworzony model wreszcie pokazał satysfakcjonujące rezultaty - poruszał się bezkolizyjnie zarówno na mapie treningowej, jak i na mapach weryfikujących i przy zmianach kierunku.

3. KOMPONENTY PROGRAMU

Główna struktura projektu składa się z **katalogu głównego** projektu, w którym znajduje się podstawowy model zawierający podstawowe zasady gry oraz pierwszą próbę napisania gry z AI. Katalog **geneticAlgorithm** zawiera pierwsze, nieudane podejście do algorytmu genetycznego (opisane szczegółowo powyżej), z kolei katalog **geneticAlgorithmLearning** jest kluczowym elementem projektu. To w nim właśnie znajduje sieć wykorzystująca algorytm genetyczny.

3.1 Katalog główny projektu

3.1.1 Map.py

Klasa **Map** w pliku **Map.py** oraz **Map2** w **Map2.py** zawiera dwie tablice z współrzędnymi, według których narysowana zostanie zewnętrzna oraz wewnętrzna linia toru.
Dane te mają następującą strukturę i wyznaczają punkty, w których linia ma zostać zakrzywiona.

```
self.out_path = [
    (3, 0),
    (1, 2),
    (0, 4),
    (0, 13),
    (2, 15),
    (4, 16),
    (12, 16),
    (16, 20),
    (25, 20),
    (25, 9),
    (20, 4),
    (19, 4),
    (15, 0),
    (3, 0)
]

self.in_path = [
    (6, 3),
    (4, 5),
    (4, 12),
    (5, 13),
    (13, 13),
    (15, 14),
    (18, 17),
    (21, 17),
    (22, 16),
    (22, 12),
    (17, 7),
    (17, 6),
    (14, 3),
    (6, 3)
]
```

W tych klasach zaimplementowane zostały również metody skalująca, wykonująca translację o dany wektor oraz rysująca linię na podstawie podanych parametrów.

3.1.2 Car.py

Kolejnym bardzo ważnym plikiem jest **Car.py**, a w nim klasa o nazwie **Car**. Tutaj przechowywane są wszystkie istotne parametry dotyczące pojazdu na torze. Oto parametry, które brane są pod uwagę podczas ruchu pojazdu:

```
self.position = Vector2(x, y)
self.speed = 2
self.velocity = Vector2(self.speed, 0.0)
self.angle = angle
self.length = length
self.sensor_angle = 80
self.sensor_angle_2 = 15
self.turning_radius = 2
```

Również w klasie **Car** posiadamy metodę, o nazwie **find_possible_intersections**, która sprawdza czy pojazd nie najechał na którąś z linii toru. Z kolei w metodzie **calculate_distance** określamy dystans w konkretnych punktach od pojazdu do ścian toru. Metoda **rotate** w opisywanej klasie definiuje sposób zmiany kierunku jazdy pojazdu. Do tego mamy jeszcze metody wyznaczające najbliższy punkt styku z przeszkodą oraz skalujące.

3.1.3 Agent.py

Klasa **Agent** w pliku **Agent.py** zawiera podstawowe parametry do stworzenia modelu zastosowanego w pierwszym podejściu do stworzenia gry z AI, która opisany został powyżej. To tutaj zdefiniowane są przede wszystkim metody budujące model z zadanymi parametrami.

```
def build_model(self):
    network = keras.Sequential()
    network.add(Dense(4000, activation='relu', input_shape=(5,)))
    network.add(Dense(3, activation='softmax'))
    optimizer = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False)
    network.compile(optimizer=optimizer,
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
    return network
```

Ponadto w metodzie **get_choice** znaleźć można zwrócone predykcje dotyczące tego czy pojazd powinien skręcić w lewo, prawo czy jechać prosto.

3.1.4 Geometry.py

Plik zawiera dwie metody odpowiadające za obliczenia wywodzące się z geometrii analitycznej. Jedna funkcja tworzy nam parametry równania linii, a druga znajduje punkt przecięcia 2 takich linii.

```python
def line(p1, p2):
    A = (p1[1] - p2[1])
    B = (p2[0] - p1[0])
    C = (p1[0] * p2[1] - p2[0] * p1[1])
    return A, B, -C


def line_intersection(L1, L2):
    D = L1[0] * L2[1] - L1[1] * L2[0]
    Dx = L1[2] * L2[1] - L1[1] * L2[2]
    Dy = L1[0] * L2[2] - L1[2] * L2[0]

    if D != 0:
        x = Dx / D
        y = Dy / D
        return x, y
```

3.1.5 Game.py

To właśnie dzięki temu plikowi oraz zawartej w nim funkcję **main** jesteśmy w stanie uruchomić podstawową wersję gry, w której to użytkownik decyduje o kierunku jazdy samochodu sterując strzałkami. Tutaj również znajduje się obsługa akcji wykonanych przez użytkownika oraz reakcji pojazdu na nie.

3.1.6 Notatnik Model Creation.ipynb

W tym pliku zawarty jest opis oraz zapisane krok po kroku budowanie sieci neuronowej oraz rezultaty działania wszystkich koniecznych do tego funkcji. Tutaj też budowany jest model sieci oraz zawarta jest metoda rysująca wyniki rezultatów. Dodatkowo zbiory danych dzielone są na testowy i treningowy i na nich odbywają się wszystkie operacje predykcji kierunku.

3.1.7 GameWithAI.py oraz GameWithAIHistory.py

Analogicznie do pliku **Game.py** i w tym przypadku oba pliki zawierają klasę **Game** oraz funkcję main umożliwiającą uruchomienie danej wybranej opcji. To znaczy uruchomienie gry z zastosowanym podstawowym AI oraz z zastosowanym podstawowym IA i dodatkowym zapisywaniem wyników do historii.

3.2 Katalog geneticAlgorithm

3.2.1 Car.py i Map.py

Oba pliki spełniają takie samo zadanie oraz posiadają taką samą strukturę jak w przypadku opisanym w sekcji powyżej.

3.2.2 NetworkIndividual.py

Zawarta w pliku klasa **NetworkUnit** pozwala na dynamiczne zainicjalizowanie obiektu o dowolnej strukturze bez względu na wymiary, które dzięki zaimplementowanym metodom są pod kontrolą.
W klasie znajdują się metody odpowiedzialne za preprocesing danych oraz wytrenowanie ich i zbudowanie modelu.

3.2.3 GAstructure.py

Tutaj zaimplementowany został nieskuteczny algorytm genetyczny, jego celem było modyfikowanie parametrów modelu oraz odpowiednia zmiana architektury sieci uczonej przez backward propagation. W metodzie ewolucyjnej wybierane są jednostki z najlepszymi wynikami jako rodzice dla kolejnych pokoleń oraz mutowane są ich “geny”, aby zachować różnorodność wśród tych najlepszych jednostek często wybierana jest jedna, w której zmieniany jest parametr, aby zapewnić swego rodzaju różnorodność i wprowadzić element losowości.
Mutacja odbywa się w następujący sposób:

```
def mutate_selected(to_mutate, parents):
    print("Mutation!")
    for idx in to_mutate:
        original_individual = parents[idx]
        original_params = original_individual.params
        param_to_mutate = random_choice(list(original_params.keys()))
        additional_param = None

        if param_to_mutate == 'lstm_neurons':
            additional_param = 'lstm_layers_number'
        elif param_to_mutate == 'lstm_layers_number':
            additional_param = 'lstm_neurons'
        elif param_to_mutate == 'dense_layers_number':
            additional_param = 'dense_neurons'
        elif param_to_mutate == 'dense_neurons':
            additional_param = 'dense_layers_number'

        new_params = original_params.copy()
        del new_params[param_to_mutate]
        if additional_param:
            del new_params[additional_param]
            #parametry nieprzekazane są dobierane losowo
        parents[idx] = NetworkUnit(params=new_params)

    print("Mutation finished!")
```

Znajdują się tutaj również metody odpowiedzialne za generowanie populacji z zadanymi parametrami oraz zapisujące historię do pliku.

3.2.4 Evaluator.py

W pliku zaimplementowany jest ruch pojazdu po mapie w konkretnym kierunku oraz metoda, która bierze pod uwagę tylko przejazdy zakończone sukcesem zarówno w prawo jak i w lewo, a następnie sumuje te dwa przypadki i zwraca jako wynik.

3.2.5 history.txt

Do pliku tekstowego history zapisywane są wszystkie rezultaty dla poszczególnych modeli razem z opisem danego modelu.

3.3 Katalog geneticAlgorithmLearning

3.3.1 Car.py i Map.py

Oba pliki spełniają takie samo zadanie oraz posiadają taką samą strukturę jak w przypadku opisanym w sekcji powyżej.

3.3.2 DummyNetwork.py

W tym pliku obsłużone zostało w zasadzie całe zachowanie sieci neuronowej. Niemal cała algebra napisana została od zera bez wykorzystania gotowego frameworka oraz bez backward propagation.
Podejście do problemu zostało zupełnie zmienione w porównaniu do poprzednich technik. Tym razem struktura sieci jest znacznie mniej skomplikowana i wygląda następująco.

```
self.nn_architecture = [
    {"input_dim": 5, "output_dim": 4, "activation": "relu"},
    {"input_dim": 4, "output_dim": 3, "activation": "sigmoid"},
]
```

Znajduje się również metoda implementująca **full_forward_propagation**

```
def full_forward_propagation(self, X):
    A_curr = X

    for idx, layer in enumerate(self.nn_architecture):
        layer_idx = idx + 1
        A_prev = A_curr

        activ_function_curr = layer["activation"]
        W_curr = self.params_values["W" + str(layer_idx)]
        b_curr = self.params_values["b" + str(layer_idx)]
        A_curr = self.single_layer_forward_propagation(A_prev, W_curr, b_curr, activ_function_curr)

    return A_curr
```

Dokładny opis działania zamieszczony jest powyżej, ale warto wspomnieć, że właśnie w tej klasie została zaimplementowane nowe podejście, czyli algorytm genetyczny zamiast modyfikować architekturę sieci, zmienia wagi w jej wnętrzu.

3.3.3 evaluator_new.py

W tym pliku można znaleźć metodę **evaluate_agent_map**, w niej podejmowana jest decyzja o kierunku jazdy oraz zaimplementowana obsługa wszelkich zdarzeń związanych z ruchem pojazdu.
W funkcji **evaluate_agent** pojazd zostaje puszczony po tej samej mapie w lewo oraz w prawo, a wyniki następnie są dodawane do siebie. Zwracana jest lipca skutecznych podejść do mapy w obu przypadkach, to znaczy pojazdy, które nie zderzyły się z granicami toru podczas jazdy w lewo i prawo.

```
def evaluate_agent(self):
    # returns number of frames 'survived' of map1 lef and right
    left_score = self.evaluate_agent_map1(direction='left')
    right_score = self.evaluate_agent_map1(direction='right')
    return left_score + right_score
```

3.3.4 GAStructureLearning.py

W pliku znajduje się klasa **GeneticAlgorithm**, która zajmuje się pełną obsługą algorytmu genetycznego. W tym znajdują się między innymi metody określające sposób inicjalizacji oraz mutowania kolejnych modeli.

```
def mutate_selected(to_mutate, parents):
    print("Mutation!")
    for idx in to_mutate:
        parents[idx].init_layers()
```

są również metody odpowiadające za: tworzenie nowej populacji na podstawie wybranych parametrów, zapisywanie wyników w odpowiednim formacie do historii oraz ewolucji. W tej ostatniej wybierane są jednostki z najlepszymi wynikami, aby później mogły zostać wykorzystane jako rodzice dla kolejnych pokoleń, dokonywane są również mutacje oraz ocena jednostek.
W metodzie zajmującej się rozmnażaniem kolejnych pokoleń waga potomka jest średnią arytmetyczną wag ich rodziców, a ewaluacja sieci pozostaje bez zmian po takim działaniu.

```python
def breed_parents(male, female):
    m_params = male.params_values
    f_params = female.params_values
    l_rate = 0.01
    new_params = {}

    for key in m_params:
        new_params[key] = []
        for i in range(len(m_params[key])):
            row = []
            for k in range(len(m_params[key][i])):
                row.append((m_params[key][i][k] + f_params[key][i][k]) / 2)
            new_params[key].append(np.array(row))

    child = DummyNetwork(new_params)

    return child
```

3.3.5 TestingBestModel.py

W tym pliku uruchomiony zostaje najlepszy model w celach wizualizacyjnych.

```python
parameters = {'W1': [np.array([-0.00812508, -0.03378669, -0.03127649, -0.15404809,  0.02644651]), np.array([-0.00439364, -0.07489187,  0.00883018, -0.01661783, -0.08608299]),
                     np.array([-0.00700202,  0.0734087 , -0.11282561,  0.00804889,  0.02363297]), np.array([-0.07046699, -0.05914622,  0.02251472, -0.11319373,  0.03472001])],
              'b1': [np.array([-0.08923098]), np.array([0.10557253]), np.array([0.23204505]), np.array([-0.08572153])],
              'W2': [np.array([-0.05361863, -0.07169517,  0.10381562, -0.12504763]), np.array([-0.04775045, -0.08695348, -0.02014614, -0.10180024]),
                     np.array([-0.08228873, -0.08303805,  0.05497919, -0.03482891])],
              'b2': [np.array([-0.01179699]), np.array([0.01515355]), np.array([-0.0748028])]}

model = DummyNetwork(parameters)

eval = Evaluator(agent=model)

eval.evaluate_agent(direction='right')
```

3.3.6 history.txt

Do pliku tekstowego history zapisywane są wszystkie rezultaty dla poszczególnych modeli razem z opisem danego modelu.