Streszczenie

Celem pracy było stworzenie oprogramowania umożliwiającego sterowanie walką robotów minisumo przy wykorzystaniu sieci neuronowej. Wagi w owej sieci miały być wyznaczane iteracyjne poprzez algorytm ewolucyjny z elementami dekompozycji. Na początku wyznaczono cele do osiągnięcia oraz zakres pracy. Przygotowano również projekt implementacji algorytmu ewolucyjnego z elementami dekompozycji oraz koncept optymalizatora wykorzystującego zdefiniowaną wcześniej metodę. Kolejnym krokiem było przygotowanie wymagań funkcjonalnych i niefunkcjonalnych dla realizowanego projektu. Potem zaprojektowano program - jego elementy składowe, wykorzystane technologie, przypadki użycia i architekturę. Ostatnim krokiem było przygotowanie odpowiednich implementacji i dokumentacji projektowej. W ostatniej części pracy oceniono wykonany system, przedstawiono jego perspektywy rozwoju, przygotowano podsumowanie i wyciągnięto odpowiednie wnioski.

Abstract

The thesis' focused on the implementation of the solution, that could be able to simulate a minisumo robots fights. The solution was required to implement a neural network driven driver allowing robot to react autonomously on the changing conditions within the dohjo. The neural network was supposed to be optimized by the evolutionary algorithm with elements of decomposition. First and foremost, we specified the main objects of the project. The following chapter covered the requirements that were put on the solution. Within the third and the fourth parts both evolutionary algorithm, and optimizer concepts were described. Across the next chapter we specified use cases and architecture of the program. Furthermore, the next part focused on the technologies that have been used. The seventh section covered implementation. The very last chapter was the place where we explained how to port C++ code to Rust programs. Afterward, we presented the summary where projects' perspectives, self-evaluation and conclusions where discussed.

Spis treści

Wstęp

Japonia jest krajem znanym z zamiłowania zarówno do tradycji, jak i nowoczesnych technologii. Ich obecna kultura jest wynikiem połączenia obu tych aspektów. Co więcej, rozwijający się dzisiaj globalizm spodował, że świat zachodni coraz mocniej korzysta z zdobyczy kulturowych krajów dalekiego wschodu. Jednym z najbardziej rozpoznawalnych elementów tradycji japońskiej są walki sumo. Ta kultywowana od VIII wieku dziedzina sportu jest od wielu lat szeroko rozpoznawana na całym świecie. W latach 80. dwudziestego wieku pojawiła się jej nowa odmiana - walki robotów minisumo.

Każdego roku setki inżynierów konstruuje nowe roboty i część z nich staje do walki w ramach corocznych mistrzostw *All Japan Robot Sumo Tournament*. Dla każdego uczestnika budowa i zaprogramowanie robota jest wyzwaniem umożliwiającym rozwój na wielu płaszczyznach. Niejednokrotnie zdarza się, że dzięki lepszym algorytmom sterującym robot potencjalnie słabszy wygrywa walki.

Zasady walk minisumo są niezwykle podobne do tych znanych z faktycznych walk sumo. Na okrągłej arenie, zwanej *dohjo* ustawiane są naprzeciwko siebie dwa roboty. Każdy z nich musi spełniać określone warunki - waga robota nie może przekraczać 0.5 kilograma, a szerokość i długość są ograniczone do 10cm. Wysokość robota, ilość sensorów i ich rodzaj są nieokreślone. Najczęściej wykorzystywane są sensory dystansowe oraz sensory naziemne. Te pierwsze służą do wykrywania przeszkód - w tym robota przeciwnika. Drugi rodzaj czujników jest stosowany w celu detekcji krawędzi ringu.

Większość algorytmów sterujących robotami wykorzystuje bardzo prymitywne mechaniki. Są to najczęściej maszyny stanów, lub proste programy oparte na pętlach i instrukcjach warunkowych. Czasem, choć o wiele rzadziej, inżynierowie odpowiedzialni za oprogramowanie algorytmu sterującego wykorzystują bardziej rozbudowane techniki - metaheurystyki, albo metody wykorzystujące sztuczną inteligencję (*SI*). Niniejsza praca w korzysta z metody posiadającej w sobie oba rozwiązania. Opracowane rozwiązanie wykorzystuje sieć neuronową, której wagi są ustalane przez metodę ewolucyjną. Dodatkowo, metoda ewolucyjna korzysta z dekompozycji.

Schemat działania stworzonej jest stosunkowo prosty. Każda *ewaluacja sieci* (iteracja) jest wykonywana dla jednej sekundy symulacji. Na podstawie danych wejściowych jakimi są zarówno historyczne, jak i obecne wartości prędkości motorów oraz stanów czujników, tworzona jest warstwa wejściowa do sieci. Następnie ewaluowane są wartości wyjściowe - prędkości motorów.

Poniższy dokument rozpoczyna się od przedstawienia celu i zakresu omawianego projektu w ramach **Rozdziału 1**. **Rozdział 2** opisuje wymagania jakie zostały postawione wobec aplikacji. **Rozdział 3** opisuje metodę *SALTGA* będącą implementacją algorytmu ewolucyjnego z dekompozycją. Kolejny z rozdziałów zajmuje się opisaniem zaimplementowanego optymalizatora. Dale przedstawiane są projekt i architektura oprogramowania. W **Rozdziale 6** prezentowany jest stos technologiczny. Następnie omawiana jest implementacja rozwiązania. Ostatnią rzeczą, przed podsumowaniem i zakończeniem jest omówienie metod wykorzystywanych przy integrowaniu (portowaniu) bibliotek napisanych w języku *C++* na język *rust*.

Praca jest opisem systemu powstałego podczas przedsięwzięcia realizowanego przez trzyosobowy zespół. Składał się on z Tomasz Durdy, Radosława Kuczbańskiego i Stanisława Markowskiego. Praca każdego z członków skupiała się w głównej mierze nad oddzielnym elementem systemu. Choć poniższa praca koncentruje się na elementach wykonanych przez autora, to dla pełnego i ogólnego opisu systemu konieczne jest odwoływanie się do komponentów przygotowanych przez innych współautorów. Do każdego odwołania zostanie dodana odpowiednia adnotacja informująca o autorze komponentu.

1 Cel i zakres pracy

Celem pracy było zaprojektowanie i wytworzenie oprogramowania umożliwiającego sterowanie robotem minisumo wewnątrz przygotowanego symulatora. Program miał wykorzystywać do tego celu sieć neuronową. Wagi w tejże sieci miały być optymalizowane i wyznaczane poprzez algorytm ewolucyjny z elementami dekompozycji. Pierwszym celem pobocznym pracy było wytworzenie oprogramowania we współdziałających ze sobą środowiskach `C++` oraz `rust`. Kolejnym celem dodatkowym było przygotowanie działającego środowiska CI/CD. Oba te cele zostały szczegółowo wydzielone i zostaną oddzielnie opisane i scharakteryzowane.

1.1 Środowisko programistyczne

Środowisko programistyczne powinno obsługiwać wiele typów systemów operacyjnych. Dodatkowo powinno ono umożliwiać programistom płynne i nieprzerwane dostarczanie zmian na odpowiednie środowiska aplikacji. Pożądaną funkcjonalnością jest też łatwe wersjonowanie i przywracanie zmian w razie potrzeby. Narzędzia dostarczone w ramach środowiska muszą być łatwe w obsłudze i późniejszym utrzymaniu.

1.2 Symulator wraz z silnikiem fizycznym

Trening robota jest elementem bardzo czasochłonnym. Aby zminimalizować nakład czasowy potrzebny do wytrenowania sieci sterującej należy zaprogramować symulator komputerowy. Musi on być wydajny obliczeniowo, a każda symulacja powinna jak najwierniej odzwierciedlać realne walki. Aby to osiągnąć symulator musi być zamodelowany w taki sposób aby każdy z robotów posiadał fizyczne atrybuty, charakterystyczne dla tej dziedziny. Silnik fizyczny musi obsługiwać masę i wymiary robotów, oraz dokładnie symulować wszelkie odbicia. Same roboty i czujniki muszą być zamodelowane w taki sposób, aby aktywacja czujników i reakcja na zmiany stanów odbywała się w odpowiednim momencie. Żaden z robotów nie powinien mieć przekazywanej informacji o stanie samego symulatora. Symulator musi być w stanie opisać stan walki poprzez jeden ze stanów - wygrana, przegrana lub remis - dla konkretnego robota.

1.3 Sieć neuronowa

Zaimplementowany algorytm sterujący robotem wykorzystuje sieć neuronową. Sieci neuronowe to popularny w ostatnich dekadach temat. Ich popularność gwałtownie wzrosła na przełomie lat 2010-2015. Widać to choćby w liczbie osób uczestniczących w głównych konferencjach na ich temat - liczba ta jest obecnie niemal piętnastokrotnie większa, niż w latach 2000-2010[1]. Zasada ich działania jest zainspirowana rzeczywistymi procesami biologicznymi zachodzącymi w systemach nerwowych zwierząt i ludzi. Niezwykła skuteczność w rozpoznawaniu wzorców oraz uniwersalność powodują, że są one doskonałym narzędziem do problemów typu blackbox i graybox, w których nieznana jest dokładna specyfika problemu.

W ramach stworzonego programu wejściami do sieci są prędkości motorów i binarne stany czujników w czasie ostatnich kilku sekund. Wyjścia natomiast reprezentują prędkości motorów robota w następnej sekundzie.

[1] https://www.skynettoday.com/overviews/neural-net-history

1.4 Sterownik robota

Roboty minisumo muszą zupełnie autonomicznie poruszać się po ringu. W tym celu stosuje się specjalne sterowniki, zaprogramowane wcześniej przez osoby tworzące roboty. Sterowniki przy wykorzystaniu najróżniejszych metod reagują na zmiany sensorów, a następnie sterują motorami i innymi elementami ruchomymi. Ich zasada działania może opierać się o proste instrukcje warunkowe, korzystać z bardziej zaawansowanych maszyn stanu, albo wykorzystywać niedeterministyczne metody sterujące jak np. algorytmy ewolucyjne, czy sieci neuronowe.

W ramach pracy stworzono sterownik wykorzystujący sztuczną inteligencję. Dzięki zaimplementowanej sieci neuronowej robot jest w stanie samodzielnie reagować na zmiany zachodzące w otoczeniu. Sterownik do optymalizacji sieci neuronowej korzysta z metody z rodziny algorytmów ewolucyjnych. Metodę można znaleźć w aplikacji pod nazwą `saltga`

1.5 Komunikacja pomiędzy komponentami w ramach stworzonego oprogramowania

Założenia wytworzonego oprogramowania spowodowały, że program musiał zostać podzielony na kilka komponentów. Skutkiem tej decyzji było obmyślenie, zamodelowanie i implementacja metod komunikacji pomiędzy wydzielonymi komponentami. Efektem ubocznym tej decyzji są zwiększenie elastyczności architektury, oraz możliwość podziału systemu na komponenty o różnym zestawie technologicznym

1.6 Trening robota

Wybór metod korzystających z sieci neuronowych implikuje przymus wytrenowania sieci. W takiej sytuacji należało opracować takie funkcjonalności które umożliwiłyby rozpoczęcie sparametryzowanego treningu oraz walidację jego skuteczności.

2 Wymagania funkcjonale i niefunkcjonalne programu

Na podstawie przedstawionych założeń i celu programu zdefiniowano następujące wymagania funkcjonalne i niefunkcjonalne.

2.1 Wymagania funkcjonalne

- Dostarczenie działającego programu udostępniającego interfejs programistyczny do rozpoczęcia, monitorowania stanu, zakończenia treningu
- Udostępnienie użytkownikowi końcowemu możliwości konfiguracji parametrów treningu - jego rodzaju, parametrów optymalizatora, parametrów fizycznych robota, parametrów dotyczących symulacji
- Dostarczenie funkcjonalności umożliwiającej uruchomienie kilku treningów jednocześnie
- Utworzenie graficznego interfejsu dla stworzonego oprogramowania
- Umożliwienie rozpoczęcia i przerwania treningu w dowolnym momencie
- Udostępnienie wag sieci dla najlepszego z wytrenowanych modeli

2.2 Wymagania niefunkcjonalne

- Wydzielenie symulatora z optymalizatorem od GUI
- Przygotowanie narzędzi dla łatwego rozwoju aplikacji w przyszłości

- Udostępnienie aplikacji poprzez stronę internetową
- Wykorzystanie języka `C++` i znajdujących się tam bibliotek w ramach stworzonego oprogramowania

3 Algorytm ewolucyjny z elementami dekompozycji

W ramach pracy należało zaimplementować *algorytm ewolucyjny* z *elementami dekompozycji*. Algorytmy ewolucyjne jest to podgrupa *metaheurystyk* zainspirowana procesami zachodzącymi w naturze.

Działanie ewolucyjnych jest ściśle związane z pojęciami *ewaluacji*, *genotypu*, *osobnika*, *populacji*, *krzyżowania*, *mutacji*, *dekompozycji*, oraz *turnieju*. Wytłumaczenie każdego z terminów jest dostępne w **podrozdziale 3.1**, gdzie znajduje się słownik pojęć związanych z tym działem metaheurystyk.

W następnych częściach pracy zastosowana metoda będzie oznaczana i nazywana przez akronim `SALTGA` - jest to skrót od nazwy `Selective Approach Linkage Tree Genetic Algorithm`.

3.1 Słownik pojęć

- *osobnik* jest pojedynczym rozwiązaniem optymalizowanego problemu. Osobnik ma w sobie zakodowaną informację o metodzie ewaluacji (taka metoda jest nazywana *ewaluatorem*)
- *fitness* jest to miara przystosowania osobnika. Informuje o jego jakości
- *genotyp* jest to zakodowana informacja o cechach osobnika. Pojedyncza wartość w genotypie jest nazywana *genem*. Przykładem genotypu dla osobnika człowieka może być ciąg liczb rzeczywistych `[1.73, 73.0, 23]`, gdzie kolejne wartości (geny) oznaczają odpowiednio wzrost w metrach, wagę w kilogramach, wiek w latach ziemskich.
- *ewaluacją* nazywamy operację wyznaczenia wartości fitnessu dla konkretnego genotypu
- *krzyżowanie* jest to operacja łączenia genotypów dwóch osobników zwanych *rodzicami* (w tradycyjnych algorytmach ewolucyjnych nie ma rozróżnienia płci).
- *mutacja* jest to operacja modyfikacji pojedynczego genotypu. Zazwyczaj genotype są mutowane w sposób zupełnie losowy
- *dekompozycja* to operacja rozbicia problemu na mniejsze części.
- *turniejem* nazwana jest operacja wybrania najlepszego osobników wśród ustalonej grupy
- *populacja* to zbiór osobników

3.2 Reprezentacja matmatyczna

Formalnie każdy osobnik jest reprezentowany poprzez ciąg liczb rzeczywistych (bądź wartości binarnych konwertowanych do liczb rzeczywistych).

$$genotype : \{x | x \in \mathbb{R}\}$$

Geny w ramach genotypu są uporządkowane, a ich kolejność ma znaczenie.

Proces wyznaczania maski (dla prezentowanego algorytmu):

$$mask\{y | y \in \mathbb{N} \setminus \{0\} \wedge y \leq |genotype|\}$$

W ogólności operacja mutacji polega na wybraniu podzbioru genów z genotypu, a następnie na modyfikacji każdego elementu z tego podzbioru

$$mutated\_genotype : \{z_j | j \notin index \Rightarrow z_j = x_j\}$$

Zakładając, że drugi osobnik jest zdefiniowany przez

$$genotype2 : \{v | v \in \mathbb{R}\}$$

Oraz zakładając warunek, że $|genotype2| = |genotype|$, operacją krzyżowania genotypów $genotype$, z $genotype2$ nazywamy poniższe działanie

$$crossed\_genotype : \{q_i | (i \in index \iff q_i = x_i) \wedge (i \notin index \iff q_i = y_i)\}$$

Populacja jest określona jako zbiór genotypów. Oznaczmy ją przez $\mathbb{P}$. Niech pojedynczy osobik o indeksie r będzie oznaczony przez p^r. Trzeba zaznaczyć, że nie jest to operacja podnoszenia do potęgi. Gen g w takim osobniku niech będzie oznaczony przez p^r_g

Operacja wyłonienia zwycięzcy z turnieju jest zdefiniowana następująco

$$T \subseteq \mathbb{P}$$

$$tournament\_winner : \exists_{t \in T} \forall_{l \in T \wedge l \neq t} evaluate(t) \geq evaluate(l)$$

3.3 Pseudokody

Pseudokod 1: Iteracja SALTGA

```
   /* Data init                                                    */
 1 masks;
 2 crossover_rate;
 3 mutation_rate;
 4 masks_regeneration_freq;
 5 tournament_size;
 6 single_gene_mutation_rate;
 7 maximum_cluster_size;
 8 genotype_size;
 9 masks_number;
   /* Masks generation                                            */
10 if iteration%masks_regeneration_freq == 0 then
11     regenerate_masks(
12         ref masks,
13         maximum_cluster_size,
14         genotype_size,
15         masks_number
16     )
   /* Crossover                                                   */
17 crossover(
18     ref masks,
19     crossover_rate,
20     tournament_size,
21 );
22 get_best(masks);
   /* Mutation                                                    */
23 foreach individual ∈ population do
24     if random_float(0, 1) < mutation_rate then
25         mutate(individual, single_gene_mutation_rate);
26 get_best(masks);
27 iteration = iteration + 1;
```

Pseudokod 2: Generacja masek *masks_generation()*

```
   /* Data init                                                                    */
 1 masks;
 2 genotype_size;
   /* Reset masks                                                                  */
 3 masks = [];
   /* Generate random masks                                                        */
 4 i = 0;
 5 while i < masks_number do
 6 |   masks += [ select_random_gene_set(genotype_size, maximum_cluster_size) ];
   /* Decompose masks                                                              */
 7 result_masks = [];
 8 linkage_tree = {};
 9 foreach mask ∈ masks do
10 |   linkage_tree += find_linkage(mask);
11 |   result_masks += decompose_linkage_tree(linkage_tree);
12 return result_masks
```

Pseudokod 3: Operacja krzyżowania *crossover()*

```
   /* Data init                                                                    */
 1 masks;
 2 crossover_rate;
 3 tournament_size;
   /* Crossover                                                                    */
 4 foreach individual ∈ population do
 5 |   foreach mask ∈ masks do
 6 |   |   if random_float(0, 1) < crossover_rate then
 7 |   |   |   single_crossover(mask, refindividual, tournament_size);
```

Pseudokod 4: Operacja krzyżowania *single_crossover()*

```
   /* Data init                                                                    */
 1 mask;
 2 individual;
 3 tournament_size;
   /* Crossover operation                                                          */
 4 other_individual = select_random_other_individual(tournament_size);
 5 individual_clone = clone(individual);
 6 foreach gene_index ∈ mask do
 7 |   individual_gene_index = other_individual_gene_index;
 8 |   if evaluate(individual) < evaluate(individual_clone) then
 9 |   |   individual = individual_clone;
```

Pseudokod 5: Operacja znajdowania powiązań *find_linkage()*

```
   /* Data init                                                    */
 1 mask;
 2 thresholds;
   /* Searching for linkage                                        */
 3 mask_clone = clone(mask);
 4 foreach gene ∈ mask_clone do
 5 |   mask_clone = mask_clone \ {gene};
 6 |   foreach other_gene ∈ mask_clone do
 7 |   |   distance = calculate_distance(individual, gene, other_gene);
 8 |   |   linkage_tree_{gene,other_gene} = distance;
 9 return linkage_tree
```

Pseudokod 6: Operacja znajdowania dystansu dla osobnika *calculate_distance()*

```
   /* Data init                                                    */
 1 gene;
 2 other_gene;
 3 modification_value;
 4 epsilon;
 5 alias gene1 = gene;
 6 alias gene2 = other_gene;
   /* Calculate distance                                           */
 7 distance = 0;
 8 individual = clone(random_individual());
   /* Get any random gene that is not equal to gene or other_gene  */
 9 random_gene = random_gene(except={gene, other_gene});
   /* Save evaluated value for individual with modified gene1      */
10 individual_gene1 -= modification_value;
11 evaluated_gene1 = evaluate(individual);
   /* Save evaluated value for changed individual with modified gene1 */
12 individual_random_gene -= epsilon;
13 evaluated_changed_gene1 = evaluate(individual);
   /* Revert gene1 modification and apply gene2 modification       */
14 individual_gene1 += modification_value;
15 individual_gene2 -= modification_value;
   /* Save evaluated value for changed individual with modified gene2 */
16 evaluated_changed_gene2 = evaluate(individual);
   /* Save evaluated value for individual with modified gene2      */
17 individual_random_gene += epsilon;
18 evaluated_changed_gene2 = evaluate(individual);
   /* Calculate distances values                                   */
19 distance_unchanged = evaluated_gene1 − evaluated_gene2;
20 distance_changed = evaluated_changed_gene1 − evaluated_changed_gene2;
21 return abs(distance_initial − distance_changed_initial)
```

3.4 Opis metody SALTGA

Wykorzystywana metoda jest wariacją algorytmu genetycznego. Poza standardowymi operacjami posiada ona dodatkową cechę - każda z operacji krzyżowania jest wykonywana nie na podstawie obliczonej, a nie zupełne losowo wygenerowanej maski. Maska jest wyznaczana podczas poszukiwania *genów liniowo zależnych*. Jako, że problem pracy posiada genotypy wielkości kilku tysięcy genów, maski są liczone na podstawie wcześniej wygenerowanego, losowego podzbioru genów.

Geny są liniowo zależne w przypadku, gdy dla zmienionego osobnika modyfikacji osobnika, zmiana wartości obu genów (każdego z osobna) wpływa identycznie na ewaluację jakości osobnika. Metoda obliczania *miary zależności genów* jest zaprezentowany w **Pseudokodzie 6**. Im mniejsza wartość tej metryki, tym większe jest prawdopodobieństwo na to, że geny są liniowo zależne (przynajmniej lokalnie, dla wyznaczonej wartości modyfikacji).

Kolejną różnicą w stosunku do tradycyjnego algorytmu genetycznego jest to, że metoda `SALTGA`, po wyznaczeniu metryki dla każdej pary genów obecnej w wylosowany zbiorze genów, dokonuje jego dekompozycji. Algorytm jest identyczny do tego z metody `LTGA`[2]. Zbiór rozbijany jest na drzewo[3]. Kolejnym krokiem jest konwersja uzyskanego drzewa na listę masek.

Metoda `SALTGA` posiada następujące hiperparametry:

- Zakres zmiany, wartość modyfikacji (`modification_value`)
- Liczba osobników biorąca udział w turnieju (`tournament_size`)
- Częstotliwość mutacji (`mutation_rate` oraz `single_gene_mutation_rate`)
- Częstotliwość krzyżowania (`crossover_rate`)
- Maksymalna wielkość generowanych klastrów (`maximum_cluster_size`)
- Liczba generowanych masek w procedurze `regenerate_masks` (`masks_number`)

Schemat ogólny opracowanej metody można zobaczyć w **Pseudokodzie 1**. Schemat krzyżowania jest zaprezentowany w **Pseudokodzie 4** oraz **Pseudokodzie 3**. Metoda budowania drzewa jest pokazana w **Pseudokodzie 5**. Sposób wyznaczania masek jest ujęty w **Pseudokodzie 2**

4 Optymalizator SALTGA dla sieci neuronowej

Sterownik robota przyjął formę narzędzia wykorzystującego wytrenowaną sieć neuronową. Sieć jest skonstruowana jako wielowarstwowy perceptron. Posiada on 3 warstwy (`layers`):

- Warstwę wejściową (zmienna liczba neuronów)
- Warstwę ukrytą (stała liczba neuronów)
- Warstwę wyjściową (dokładnie 2 neurony)

Każda z neuronów w warstwie `i`-tej jest połączony z każdym z neuronów w warstwie `i+1`. Dla każdego połączenia jest zdefiniowana odpowiednia waga. Ponadto, wszystkie neurony mają zdefiniowany bias.

Sieć neuronowa obsługuje następujące funkcje aktywacji: `ReLu`, funkcję sigmoidalną, oraz funkcję tangensa hiperbolicznego. Funkcja aktywacji na ostatniej warstwie jest nazwana funkcją normalizacji (*Normalization Function*).

[2]https://link.springer.com/chapter/10.1007/978-3-642-15844-5_27

[3]https://www.researchgate.net/publication/341231450/figure/fig1/AS:888764134088710@1588909290690/An-example-of-hierarchical-clustering-algorithm-on-7-genes.ppm

4.1 Dane wejściowe sieci

Sieć przyjmuje ciąg danych rzeczywistych jako dane wejściowe. W przypadku omawianego problemu, są to wartości sensorów i prędkości motorów w poprzedniej sekundzie (poprzedniej iteracji/klatce). W celu zwiększenia dokładności i skuteczności działania robota, postanowiono o dodaniu danych historycznych do danych wejściowych. Wytrenowana w ten sposób sieć neuronowa, jest w stanie o wiele lepiej zareagować na zmiany otoczenia. Niestety, takie rozwiązanie znacząco zwiększa ilość wag i biasów. To zaś znacząco wpływa na złożoność obliczeniową problemu

Wartości sensorów są wartościami binarnymi. W celu dostosowania do formatu wejściowego sieci, należało przekonwertować wejścia na liczby rzeczywiste. Wybrano wartości -1, dla czujnika nieaktywnego, oraz 1, dla sensora pobudzonego (aktywnego).

Wartości motorów (zarówno na wejściu, jak i wyjściu) są wyrażone w postaci wartości rzeczywistych w przedziale `[-1, 1]`. W tym wypadku, `-1` oznacza pełną moc motora w kierunku przeciwnym do obecnej orientacji robota. `0` jest tożsame z brakiem mocy na silniku koła. Z kolei `1` oznacza pełną moc w kierunku zgodnym z orientacją robota.

Dane wejściowe sieci są aktualizowane na bieżąco, co 1 sekundę (w każdej iteracji).

4.2 Dane wyjściowe sieci

Na wyjściu sieci znajdują się dokładnie 2 neurony odpowiadające za prędkość w następnej iteracji symulatora.

4.3 Powód zamieszczania informacji o stanach historycznych na wejściu sieci

Brak historii oznacza możliwość reagowania tylko na stan obecny. Intuicyjnie wydaje się być oczywiste, że ucieczka robota w stronę granicy ringu w celu uniku w ostatniej chwili, jest inną sytuacją niż przypadkowe podjechanie pod linię graniczną. Aby rozróżnić te sytuacje potrzebna jest wiedza na temat stanów poprzednich. Jako, że walki są niezwykle dynamiczne[4], historia kilku iteracji wstecz jest wystarczająca.

```rust
fn fitness(max_frames: usize, simulation_log: &SimulationLog) -> f32 {
    let win = {
        let frame = simulation_log.frames.iter().last().unwrap();

        if frame.robot1.lost {
            0.0
        } else if frame.robot2.lost {
            20.0
        } else {
            5.0
        }
    };

    let time = 5. * simulation_log.frames.len() as f32 / max_frames as f32;

    win - time + 5.
}
```

Rysunek 1: Funkcja licząca fitness dla optymalizatora SALTGA

[4]https://www.youtube.com/watch?v=Ri6BwwMmCqc

4.4 Liczenie fitnessu

Kluczowym aspektem dla skuteczności programu jest prawidłowo działająca metoda licząca fitness. Z jednej strony oczywiste jest, że nie może ona premiować walk przegranch. Nie powinna też karać za wygranie walki. Jednak bardzo ważne okazuje się być rozróżnianie, czy walka została wygrana w ciągu 10s, czy w ciągu 50s. Kolejnym zagadnieniem jest przedział wartości fitnessu. Okazuje się, że wartości poniżej 0 wielokrotnie zatrzymywały potencjalnie interesującego osobnika. Jego przystosowanie, z powodu jednej prostej porażki, okazywało się mieć marginalną wartość.

Na podstawie rozmów z zespołem i doświadczeń innych jego członków ustalono, że racjonalnym będzie ustawienie takiej metody liczenia fitnessu, aby wyeliminować wszystkie z powyższych problemów. Na **Zdjęciu 1** widoczna jest obowiązująca implementacja funkcji fitnessu.

4.5 Losowanie pozycji

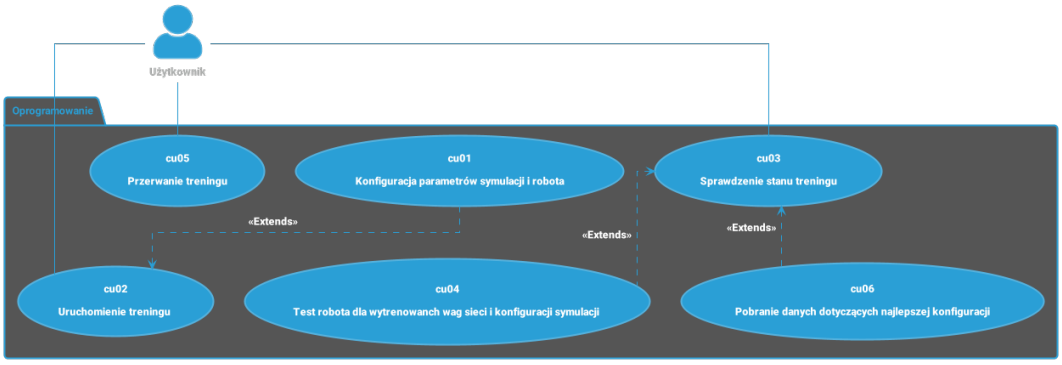
W ramach optymalizatora, można skonfigurować, czy wylosowane pozycje będą zupełnie losowe, czy roboty będą ustawiane w sposób zgodny z zasadami walk minisumo. Obie metody zostały zaimplementowane przez Radosława Kuczbańskiego.

W ramach konfiguracji symulacji, można wybrać która z metod będzie wykonywana.

5 Projekt i architektura oprogramowania

Wykorzystując podane wyżej wymagania, oraz zakres pracy stworzono projekt i koncept architektury rozwiązania. Architektura została zdefiniowana na dwóch poziomach - poziomie logicznym, oraz poziomie koncepcji systemu.

Architektura na poziomie logicznym definiuje architekturę rozwiązania z perspektywy przypadków użycia. Poziom koncepcji systemu obejmuje połączenia pomiędzy bibliotekami oraz fizycznymi programami.



Rysunek 2: Diagram przypadków użycia

5.1 Przypadki użycia

Głównym celem zaprojektowanego oprogramowania jest dostarczenie konfigurowalnego symulatora. Z tego powodu wszystkie przypadki użycia dotyczą właśnie tego komponentu.

Zasadniczą funkcjonalnością jest skonfigurowanie symulacji (`cu01`), a następnie rozpoczęcie treningu (`cu02`). W ramach tych przypadków użycia znajdują się między innymi:

Nazwa	**Wartość**
Identyfikator PU	cu01
Nazwa	Konfiguracja parametrów symulacji i robota
Aktorzy	Użytkownik
Warunki początkowe	BRAK
Przebieg	1. Użytkownik konfiguruje parametry symulacji 1a. Użytkownik konfiguruje parametry fizyczne symulacji 1b. Użytkownik konfiguruje motory i sensory 2. Użytkownik wybiera rodzaj optymalizatora 3. Użytkownik konfiguruje parametry optymalizatora

Tabela 1: Przypadek użycia cu01

- wybranie rodzaju optymalizatora
- skonfigurowanie hiperparametrów wybranego optymalizatora (algorytmu optymalizującego wagi dla sterownika operującego na sieci neuronowej)
- wybranie parametrów dla symulacji i robotów
- uruchomienie symulacji z podaną konfiguracją

W czasie treningu wymagana jest możliwość sprawdzenia stanu uczenia (`cu03`) wraz możliwością testu funkcjonalności robota dla wytrenowanej sieci (`cu04`), oraz możliwością pobrania wag dla najlepiej wytrenowanej sieci (`cu06`).

Finalnie użytkownik powinien mieć możliwość zakończenia treningu w dowolnym momencie (`cu05`)

Nazwa	**Wartość**
Identyfikator PU	cu02
Nazwa	Uruchomienie treningu
Aktorzy	Użytkownik, System
Warunki początkowe	Użytkownik przygotował konfigurację treningu
Przebieg	1. Użytkownik wysyła zapytanie do systemu o rozpoczęcie treningu dla podanej konfiguracji 2. System informuje użytkownika o statusie treningu wraz z informacją o ID
Przebieg alternatywny	1. Użytkownik wysyła zapytanie do systemu o rozpoczęcie treningu dla podanej wadliwej konfiguracji 3. System informuje użytkownika o błędzie

Tabela 2: Przypadek użycia cu02

Nazwa	**Wartość**
Identyfikator PU	cu03
Nazwa	Sprawdzenie stanu treningu
Aktorzy	Użytkownik, System
Warunki początkowe	1. Przynajmniej jeden trening został uruchomiony 2. Wysyłane ID treningu jest poprawne
Przebieg	1. Użytkownik wysyła zapytanie do systemu o stan treningu o danym ID 2. System wysyła informację zwrotną z informacją o treningu

Tabela 3: Przypadek użycia cu03

Nazwa	**Wartość**
Identyfikator PU	cu04
Nazwa	Test robota dla wytrenowanch wag sieci i konfiguracji symulacji
Aktorzy	Użytkownik, System
Warunki początkowe	Wysyłana konfiguracja jest poprawna
Przebieg	1. Użytkownik wysyła zapytanie do serwera o uruchomienie symulacji dla zadanych parametrów 2. System uruchamia symulację dla podanych parametrów 3. System zwraca użytkownikowi informację o wyniku symulacji

Tabela 4: Przypadek użycia cu04

Nazwa	**Wartość**
Identyfikator PU	cu05
Nazwa	Przerwanie treningu
Aktorzy	Użytkownik, System
Warunki początkowe	1. Przynajmniej jeden trening został uruchomiony 2. Wysyłane ID treningu jest poprawne
Przebieg	1. Użytkownik wysyła zapytanie do systemu o zakończenie treningu z danym ID 2. System kończy trening

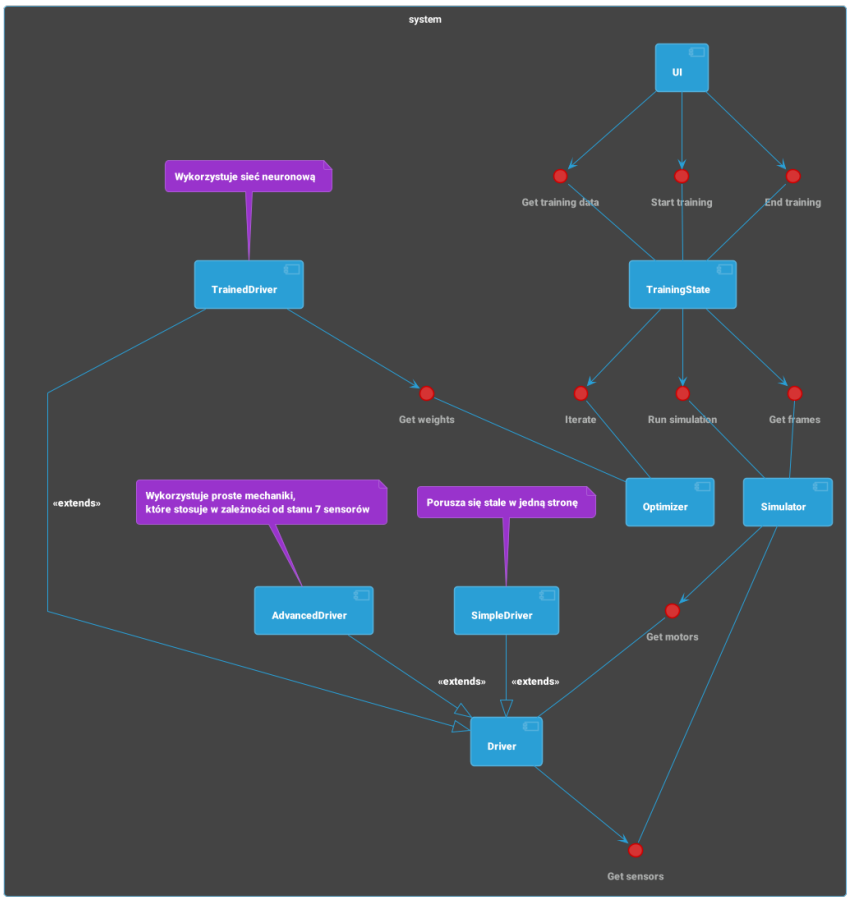
Tabela 5: Przypadek użycia cu05

Nazwa	**Wartość**
Identyfikator PU	cu06
Nazwa	Pobranie danych dotyczących najlepszej konfiguracji
Aktorzy	Użytkownik, System
Warunki początkowe	1. Przynajmniej jeden trening został uruchomiony 2. Wysyłane ID treningu jest poprawne
Przebieg	1. Użytkownik wysyła zapytanie do systemu o najlepszą konfigurację sieci 2. System odsyła wagi dla najlepszej sieci do tej pory

Tabela 6: Przypadek użycia cu06

5.2 Architektura logiczna oprogramowania

Na podstawie istniejących przypadków użycia wyszczególniono odpowiednie komponenty. W ten sposób ustalono topologię połączeń pomiędzy nimi oraz architektórę logiczną, którą tworzą. W ramach diagramu komponentów architektury logicznej wyodrębiono także logiczne interfejsy, które spełniają wymagania odpowiednich przypadków użycia



Rysunek 3: Architektura logiczna

Nazwa	Krótki opis	Typ
UI	Interfejs użytkownika. Zarówno GUI, jak i API	K
TrainingState	Komponent zarządzający treningiem	K
Simulator	Uruchamia i przeprowadza obliczenia fizyczne	K
Driver	Steruje robotem wewnątrz symulacji	K
Optimizer	Ustala wagi w ramach treningu	K
Start training	Użytkownik uruchamia trening sieci	I
End training	Użytkownik kończy trening sieci	I
Get training data	Użytkownik pobiera dane na temat treningu sieci	I
Get frames	Pobór katek do statystyk dotyczących treningu	I
Run simulation	Uruchomienie symulacji poprzez inny komponent	I
Iterate	Wykonananie iteracji przez optymalizator	I
Get motors	Uzyskanie prędkości motorów dla następnej sekundy	I
Get sensors	Uzyskanie listy sensorów dla sterownika	I
Get weights	Pobieranie wag dla sterownika sterującego robotem	I

Tabela 7: Wyodrębnione komponenty i ich interfejsy dla architektury logicznej. K - komponent, I - interfejs

5.3 Architektura na poziomie koncepcji systemu

Wybrana architektura systemu wymagała wielu różnych funkcjonalności. Z tego powodu aplikację podzielono na oddzielne komponenty w ramach architektury koncepcji systemu. Połączenia pomiędzy komponentami również zostały zdefiniowane.

Oprogramowanie składa się z dwóch programów - `frontend` (interfejs graficzny) oraz `backend` (aplikacja serwerowa). Ponadto, projekt `backend` składa się z części właściwej, oraz bibliotek pomocniczych:

- `neural_net` (napisana w języku `rust`). Służy do obsługi sieci neuronowej. Biblioteka powstała z powodu problemów technicznych związanych z dostępnością głównego programu do wag sieci.
- `decomposition` (napisanej w języku `C++`). Służy do obsługi drzewa (*linkage tree*) powiązań w ramach metody `saltga` opisanej szczegółowo w **Rozdziale ??**

5.3.1 Uzasadnienie wyboru architektury

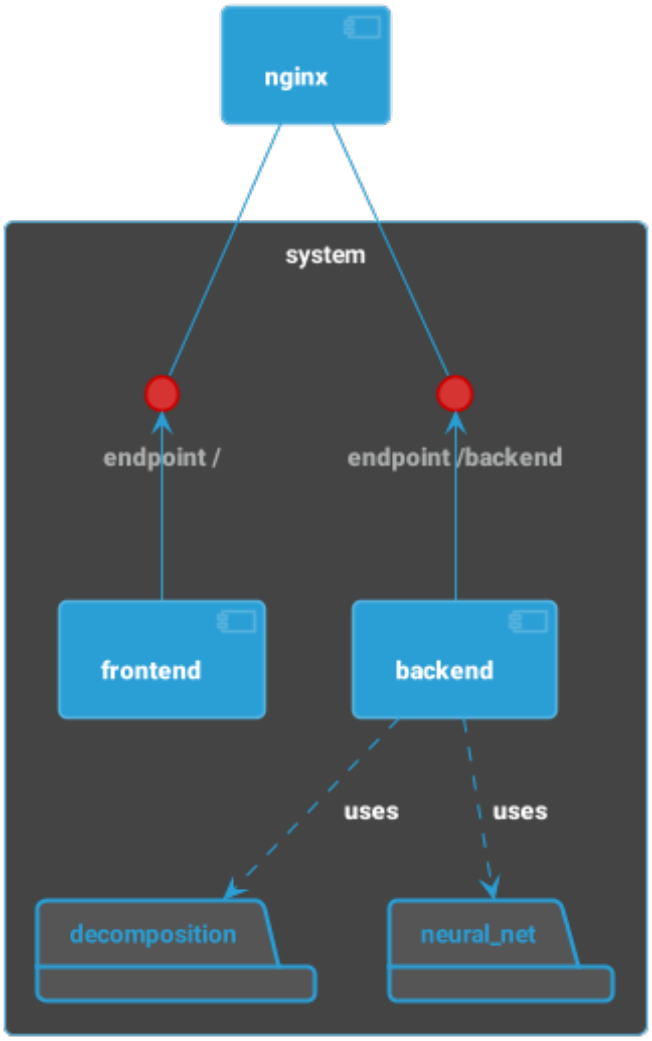
Ponieważ funkcjonalność sieci neuronowej jest stosunkowo szerokim zagadnieniem postanowiono ją wydzielić jako oddzielną bibliotekę. Na początku rozważano przeniesienie sieci wraz z optymalizatorami do oddzielnej aplikacji, jednak zaniechano tego pomysłu. Perspektywa obsługi, optymalizacji i późniejszego utrzymania kanałów komunikacji pomiędzy optymalizatorem z siecią, a silnikiem fizycznym z symulatorem wydawała się nieprawdopodobna do realizacji. Z tego powodu wydzielono bibliotekę `neural_net`

Wydzielenie backendu i frontendu było spowodowane chęcią uproszczenia struktury kodu. Na korzyść tego rozwiązania wpłynęła także możliwość prostego, równoległego utrzymywania i rozwoju obu tych komponentów.

Wydzielenie biblioteki do dekompozycji było realizacją jednego z celów pobocznych tej pracy - integracją `C++` oraz `rust`. Wybór ten znacząco wpłynął na prędkość realizacji poniższej pracy.

5.3.2 Inne komponenty

W rozwiązaniu produkcyjnym został również umieszczony komponent nginx, który służy jako serwer odwróconego proxy (*reverse proxy*) dla aplikacji.



Rysunek 4: Architektura na poziomie koncepcji systemu (rozwiązanie produkcyjne)

6 Wykorzystywane technologie

W momencie skompletowania założeń systemowych można było rozpocząć etap implementacji. Zaczęto przegląd dostępnych technologii, a następnie wyselekcjonowano zestaw w którym zrealizowano pracę. Okazało się, że wykorzystano bardzo szeroki wachlarz technologiczny. Z tego powodu, w celu opisania poszczególnych technologii postanowiono podzielić go. Wybrano podział ze względu na miejsce zastosowania konkretnego rozwiązania.

6.1 Backend oraz biblioteka `neural_net`

6.1.1 Rust

Wieloparadygmatowy język powstały w 2010 roku. Stworzony na podobieństwo `C++`, zapewniający bezpieczeństwo pamięci i podobną wydajność]. W stosunku do `C++` jest on zaledwie o 10% mniej wydajny[5].

Podczas rozważania technologii dla komponentu `backend` i biblioteki obsługującej sieci neuronowe brano pod uwagę języki charakteryzujące się wysoką wydajnością obliczeniową. Było to podyktowane dużymi nakładami obliczeniowymi potrzebnymi do nauki sieci w ramach optymalizatora. Rozważano kilka technologii: `rust`, `C++`, `C`, oraz `C#`. Od razu zrezygnowano z języka `C`. Było to spowodowane jego bardzo ubogą składnią i brakiem kluczowych bibliotek.

Pomimo tego, że `C++` spełniał warunek wydajności obliczeniowej to także został odrzucony. Jest to język wysokopoziomowy, statycznie typowany i wieloparadygmatowy. W dużej mierze jest kompatybilny z `C`. `C++` daje swobodę w zarządzaniu pamięcią, oraz posiada wiele bibliotek umożliwiających portowanie gotowych rozwiązań na inne języki. Z tego powodu jest niezwykle często wybierany jako narzędzie do implementacji algorytmów i programów wymagających zarówno elastyczności, jak i dużej wydajności. Jego główną wadą jest to, że nie jest to język bezpieczny - nie zapewnia on w momencie kompilacji bezpieczeństwa pamięci. To powoduje, że nawet najmniejszy błąd programisty, może skutkować wyciekami pamięci. Poza tym zarządzanie zależnościami, czy obsługa systemu budującego wymaga dużych nakładów czasowych.

Następną rozważaną opcją było wykorzystanie `C#`, będącego *następcą `C++`*[6]. Jest to język składniowo podobny do `Javy`. Podobnie do niej posiada on odśmiecacz (*garbage collector*) i dużą ilość gotowych bibliotek. Jednak w przeciwieństwie do `Javy`, `C#` umożliwia sterowanie pamięcią niskopoziomowo (choć nie jest to bezpieczny mechanizm)[7] oraz bardzo prosto integruje się go z bibliotekami napisanymi w języku `C++`. W czasie rozważania wzięto pod uwagę niższą wydajność `C#` oraz dostępne biblioteki, które nie spełniały wymagań projektowych. Ostatecznie zrezygnowano z tej opcji.

Finalnie zapadła decyzja o wykorzystaniu technologii `rust`. Jest to język statycznie typowany, niewiele mniej wydajny od `C++`. Pomimo bycia językiem wysokopoziomowym umożliwia on wykonywanie skomplikowanych operacji niskopoziomych. Różni się on od języka `C++` tym, że podczas kompilacji zapewnione jest bezpieczeństwo pamięci - domyślnie `rust` uniemożliwia wycieki pamięci. W przeciwieństwie do większości języków w `rust` domyślnie stosuje się mechanizmy przenoszenia pamięci (`move semantics`[8]). Mechanizm polega na tym, że zamiast kopiowania, czy przekazywania referencji, obiekty przekazują sobie prawo do korzystania z danego obiektu. To powoduje niską konsumpcję pamięci operacyjnej. Język jest stosunkowo młody, bo ma zaledwie 12 lat. Pomimo tego, jest to język z nieprawdopodobnie szeroką gamą dostępnych bibliotek. Jego składnia umożliwia szybką pracę nad kodem aplikacji.

[5] https://raw.githubusercontent.com/dmitryikh/rust-vs-cpp-bench/master/performance_table.png
[6] https://wazniak.mimuw.edu.pl/index.php?title=AWWW-1st3.6-w06.tresc-1.1-Slajd14
[7] https://www.c-sharpcorner.com/article/pointers-in-C-Sharp/
[8] https://doc.rust-lang.org/rust-by-example/scope/move.html

6.1.2 Cargo

Elastyczny system budujący(buildsystem) w ekosystemie technologii `rust`. Jest on oficjalnie wspierany przez autorów języka. Umożliwia budowanie zarówno ze scentralizowanego repozytorium, jak i repozytoriów `git`.

Naturalnym wyborem dla `rust`a było wykorzystanie buildsystemu `cargo`. Umożliwił on połączenie biblioteki napisanej i skompilowanej w języku `C++`. Efektami ubocznymi korzystania z `cargo` były również automatyczne generowanie dokumentacji projektowej na podstawie kodu, czy łatwe dołączanie testów jednostkowych.

6.1.3 Rocket.rs

Biblioteka służąca do obsługi serwera http.

Wytworzenie API dla aplikacji backendowej wymagało zastosowania specjalnej biblioteki. Dzięki swojej prostemu interfejsowi rozwój serwera webowego służącego do komunikacji z optymalizatorami i symulatorem nie wymagało dużych nakładów czasowych. Dodatkowo, dzięki bibliotekom pochodnym backend został opatrzony w graficzny interfejs (swagger) pokazujący wszystkie punkty dostępowe (*endpointy*) interfejsu programistycznego (*API*). Jest to jedna z najlepszych metod dokumentacji API, wykorzystywana w aplikacjach biznesowych od wielu lat.

6.1.4 Rapier2D

Silnik fizyczny dla języka `rust`

Kolejnym elementem który musiał być wybrany był silnik fizyczny, który byłby wykorzystywany przez symulator. Jedynym sensownym wyborem okazał się być `rapier2D`. Symulatorem zajmował się w głównej mierze Radosław Kuczbański.

6.1.5 ndarray

Biblioteka do obługi operacji na macierzach. Biblioteka jest portem jej odpowiednika w języku `C++`

W celu uproszczenia kodu, oraz zwiększenia wydajności operacji macierzowych, zdecydowano się na zastosowanie biblioteki `ndarray`. Jest to narzędzie znane głownie z `pythona` (`ndarray` jest elementem biblioteki `numpy`). Wykorzystanie tego rozwiązania umożliwiło uproszczenie kodu oraz zmniejszenie nakładów obliczeniowych.

6.1.6 Rayon

Biblioteka umożliwiająca łatwe zrównoleglanie zadań

W celu optymalizacji komponentów zdecydowano się na wykorzystanie biblioteki `rayon`. Składnia napisanego programu jest w dużej części oparta o paradygmat funkcyjny. `Rayon` umożliwił bardzo szybkie zrównoleglanie zadań. Jako, że biblioteka posiada wiele integracji, można było równoleglić także operacje macierzowe wykonywane przez `ndarray`.

6.2 Frontend

Część frontendowa przypadła Stanisławowi Markowskiemu. Z tego powodu wykorzystane technologie w ramach tej części zostaną tylko wymienione. W ramach aplikacji frontendowej wykorzystano:

- Typescript
- ReactJS
- Material Design

6.3 Dekompozycja

6.3.1 C++

Celem pobocznym tej pracy było pokazanie integracji `C++` z `rust`em. Z tego powodu biblioteka obsługująca algorytm dekompozycji (budowę `linkage tree`), została zaimplementowana w `C++` w wersji 17 (standard `C++17`).

6.3.2 CMake

Popularny buildsystem dla `C++`. Od wersji 3.10 jest to jedno z najlepszych, dostępnych na rynku rozwiązań dla cpp.

Po wyborze `C++` jako jęzka programowania należało wybrać buildsystem, który umożliwiłby kompilację, instalację i specyfikację zależności dla pisanego komponentu. Do wyboru dostępne były `make`, `CMake`, `ninja`, `scons` oraz `meson`. Wybór `CMake`a był spowodowany wcześniejszą znajomością tego buildsystemu.

6.3.3 Doxygen

Narzędzie służące go generowania dokumentacji na podstawie istniejącego kodu. Wykorzystywane głównie dla projektów napisanych w `C++`

Do wykonania dokumentacji kodu wykorzystano narzędzie `doxygen`. Dla narzędzia nie znaleziono działających, dobrze rozwiniętych i utrzymywanych alternatyw.

6.4 CI/CD

W ramach pracy nad systemem przygotowano system służący do usprawnienia pracy nad kodem aplikacji.

6.4.1 SemVer

Skrót oznacza "Semantyczne Wersjonowanie". Jest to popularna metodyka wersjonowania narzędzi. Każda wersja jest reprezentowana poprzez ciąg tekstowy w formie `{major}.{minor}.{patch}`

W celu uproszczenia pracy z innymi narzędziami deweloperskimi wykorzystano autorskie skrypty[9] wykorzystujące mechanizmy `GIT`a (`git hooks`)[10]. Efektem ubocznym wersjonowania kodu przy pomocy tej metodyki były łatwe poruszanie się po zmianach, bezproblemowe przywracanie wadliwych zmian, a także przygotowanie aplikacji do ewentualnego, późniejszego rozwoju.

6.4.2 Gitlab

Jest to otwartoźródłowa platforma integrująca w sobie zdalne repozytorium `GIT`*, narzędzia umożliwiające bezingerencyjne budowanie, testowanie i wdrażanie oprogramowania*

Zespołowy charakter projektu inżynierskiego wymusił na członkach przedsięwzięcia inżynierskiego wybór zdalnego repozytorium. Po wstępnych ustaleniach wybrano `GitLaba` ze względu na jego otwartoźródłowy charakter. Dodatkowymi czynnikami które wpłynęły na wybór tej platformy było zintegrowane środowisko DevOps. To pozwoliło zautomatyzować niektóre aspekty związane z utrzymaniem kodu, jego testowaniem, budowaniem i wdrażaniem powstałych aplikacji.

6.4.3 Docker

Zestaw narzędzi służący do konteneryzacji oprogramowania

Szeroki wachlarz stosowanych języków i technologii jest wielokrotnie powiązany z problemami związanymi z lokalnym budowaniem i odpalaniem aplikacji. W celu uniknięcia takich problemów zdecydowano się na wykorzystanie `dockera`. Niestety, ale w momencie integracji z biblioteką `decomposition` narzędzie te okazało się utrudniać pracę nad oprogramowaniem. Z powodów ograniczonych zasobów czasowych, rozwój skonteneryzowanego rozwiązania został w pewnym momencie wstrzymany. Wykorzystanie `dockera` daje jednak duże możliwości w przypadku chęci rozwoju oprogramowania wytworzonego w ramach zespołowego przedsięwzięcia inżynierskiego.

6.4.4 Nexus

Zdalne repozytorium dla plików binarnych

Przechowywanie zbudowanych obrazów dockerowych wymagało wyboru odpowiedniej platformy. Do wyboru były `dockerhub` oraz `nexus`. Finalnie postanowiono wykorzystać drugie rozwiązanie, z powodu jego uniwersalności (poza obrazami `docker`owymi, `nexus` posiada wiele innych silników służących do przetrzymywania plików binarnych). W przyszłości wybór ten może znacząco poprawić doświadczenie użytkownika dla osób rozwijających oprogramowanie

6.5 Inne

[9] https://gitlab.com/drimtim/backend/-/tree/master/.githooks
[10] https://githooks.com/

6.5.1 Nginx

Serwer webowy

Do obsługi certyfikatów SSL potrzebne było kolejne narzędzie - serwer webowy. Do wyboru były `caddy`, `traefic`, `apache httpd` oraz `nginx`. Prostota konfiguracji, oraz wcześniejsza znajomość programu wpłynęły na wybór ostatniego z wymienionych rozwiązań.

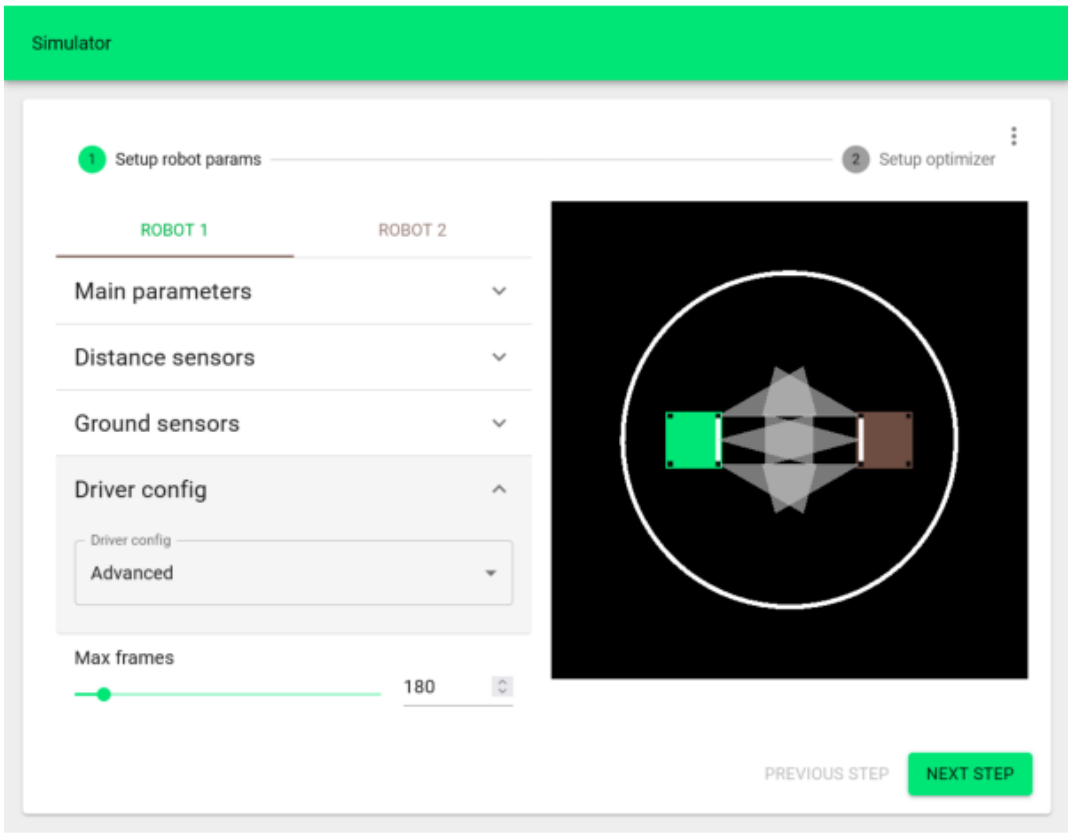
6.5.2 GIT

Zespołowy charakter pracy zmusił grupę do wyboru systemu kontroli wersji. Wykorzystano `GIT`a. Repozytoria znajdują się zarówno na `gitlab.com`, jak i `gitlab.dzordzu.pl`. Do ustawiania URLów źródeł wykorzystano prosty, autorski skrypt (`repo_configure.sh`) znajdujący się w każdym z kluczowych repozytoriów. W innych wypadkach manualnie ustalono repozytoria za pomocą komendy `git remote add`

7 Implementacja

Zaprojektowanie algorytmu, ustalenie zestawu technologii, oraz przygotowanie koncepcji architektury rozwiązania umożliwiło przejście do kluczowej fazy rozwoju aplikacji - implementacji.

Praca nad rozwiązaniem została podzielona na trzy osoby. Stanisław Markowski wykonał część obejmującą interfejs graficzny. Radosław Kuczbański przygotował symulator. Tomasz Durda zaprogramował sieć neuronową oraz podstawy sterownika obsługującego roboty sterowane za pomocą wytrenowanej, sztucznej inteligencji.



Rysunek 5: Wygląd interfejsu graficznego dla wykonanego oprogramowania

Ścieżka	Krótki opis	Metoda
/swagger-ui/	Spis wszystkich endpointów API	GET
/run	Uruchamianie symulacji na podstawie wysłanego zapytania	POST
/train	Uruchamianie treningu na podstawie wysłanego zapytania	POST
/train/{uuid}	Kończenie treningu o podanym UUID	DELETE
/train/{uuid}	Uzyskanie informacji na temat treningu o danym UUID	GET
/train/{uuid}/test	Uzyskanie klatek przykładowej symulacji dla najlepiej wytrenowanej sieci	GET
/train/{uuid}/config	Uzyskanie konfiguracji danego treningu	GET

Tabela 8: Endpointy serwera HTTP komponentu backend

7.1 Implementacja backendu

7.1.1 Sterownik robota

Jak zostało już wspomniane wcześniej, `rust` jest językiem wieloparadygmatowym. Umożliwiło to korzystanie zarówno z paradygmatu funkcyjnego, jak i paradygmatu obiektowego. Główne elementy są zaimplementowane w postaci struktur z narzuconymi interfejsami - wykorzystywane jest podejście obiektowe. Do operacjach na kolekcjach postanowiono wykorzystać gotowe, napisane funkcyjnie, rozwiązania, często będące częścią oficjalnej składni języka `rust`.

W ramach komponentu `backend` zaimplementowano sterownik robota obsługujący sieć neuronową - `TrainedDriver`. Implementuje on interfejs `DriverTrait`, w którym zdefiniowano metodę `get_motors`. Interfejs `DriverTrait` jest współdzielony dla każdego sterownika w ramach zaimplementowanego rozwiązania.

Sterownik `TrainedDriver` przyjmuje w ramach metody `get_motors` zestaw wejść z sensorów. Wejścia są w przedstawione w postaci wektora wartości binarnych. W ramach metody, wartości te są konwertowane do liczb rzeczywistych. Następnie do uzyskanej listy dodawane są dwie wartości rzeczywiste reprezentujące prędkości motorów robota, oraz dodawane są wartości historyczne danych wejściowych sieci.

`TrainedDriver` przechowuje w sobie informację o wagach sieci neuronowej. Sieć neuronowa ma zawsze identyczną, ustaloną strukturę. Metoda `get_motors` wykonuję ewaluację wartości wyjściowej, stosując utworzony wcześniej wektor (listę) jako dane wejściowe do sieci. Danymi wyjściowymi sieci są prędkości motorów, które są zwracane jako wynik metody `get_motors`.

7.1.2 Serwer HTTP

Serwer HTTP obsługuje kilka kluczowych endpointów. Ich listowanie za pomocą technologi `swagger` jest uzyskane poprzez dodanie makra `JsonSchema` do każdej z metod obsługujących endpointy HTTP (oraz do klas wykorzystywanych w ich zakresie). Jest to interfejs umożliwiający serializację schematu zapytania HTTP do postaci pliku w formacie `json`. Dodatkowo należało zaimplementować interfejs `OpenApiResponderInner` dla struktury `TrainingError`.

7.2 Implementacja sieci neuronowej

Do realizacji pracy była konieczna działająca sieć neuronowa. Choć początkowo planowane było wykorzystanie gotowych rozwiązań (takich jak `torchrs`[11]), to powstałe problemy z dostępem do wag w ramach sieci (dostępu do `varstore`) uniemożliwiły wykonanie takiego zabiegu.

W taki sposób zaimplementowana została biblioteka obsługująca strukturę, ewaluację, a nawet proste metody uczenia maszynowego sieci neuronowej.

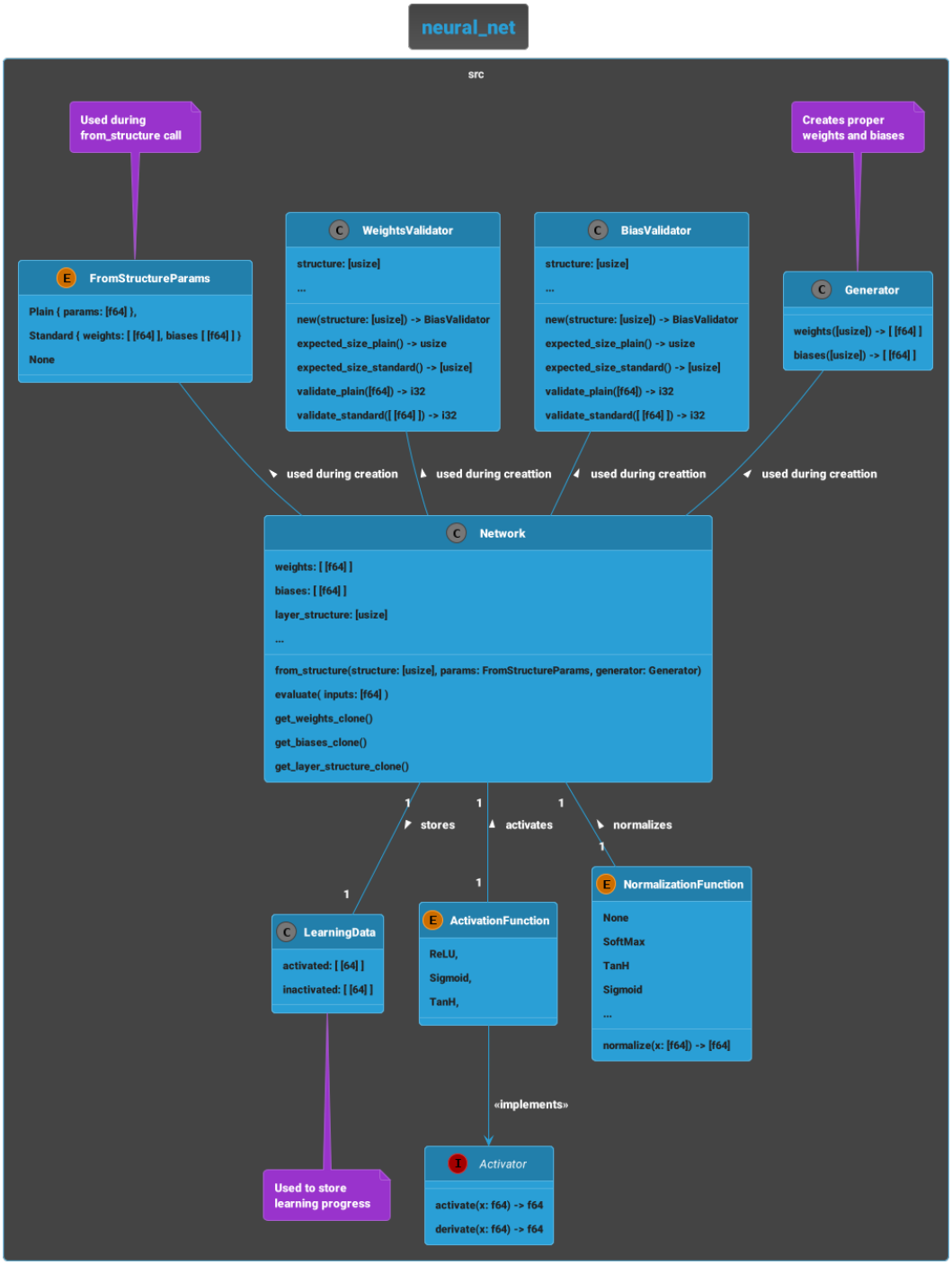
7.2.1 Testy jednostkowe

Dla zaimplementowanej sieci stworzono także zestaw testów jednostkowych sprawdzający poprawność jej działania. W ramach testów testowano:

1. generację wag początkowych i biasów dla różnych rozmiarów struktór
2. ewaluację sieci dla konkretnych przykładów
3. metody walidacji i korekcji wag

Testy wykorzystywały wbudowane mechanizmy buildsystemu `cargo`. Każda z funkcji w ramach katalogu `tests` wykorzystywała makro ustalające, że należy ona do zbioru uruchomieniowego w ramach komendy `cargo test`

[11] https://github.com/torchrs/torchrs



Rysunek 6: Diagram klas - biblioteka neural_net

Rysunek 7: Diagram reprezentujący zbiory testów jednostkowych - biblioteka neural_net

Rysunek 8: Przykładowe uruchomienie testów jednostkowych dla sieci neuronowej

7.3 Implementacja algorytmu dekompozycji

Dekompozycja wykorzystująca mechanizmy `linkage tree` została zaimplementowana w ramach oddzielnej biblioteki. Budowanie, jak i instalacja zostały obsłużone przez buildsystem `CMake`.

Struktura biblioteki jest stosunkowo prosta. Składa się ona z nasypujących klas:

1. `DisjointCluster`
2. `PossibleClustering`
3. `ClusterCalculator`

Pierwsza z klas przetrzymuje wektor trzymający klaster oraz informację o minimalnym indeksie znajdującym się w klastrze. Dodatkowo udostępnia ona operatory nierówności które umożliwiają sortowanie instancji klasy.

Druga z klas przechowuje dwa obiekty typu `DisjointCluster`. Ona również jest zaopatrzona w operatory nierówności.

Trzecia i ostatnia z zaimplementowanych klas służy do dekompozycji problemu za pomocą algorytmu identycznego jak w klasycznym `LTGA`. Klasa jest przystosowana do integracji z językiem `rust` i zapewnia wygodne metody wykorzystujące jedynie typy prymitywne. Taki zabieg znacząco ułatwia portowanie kodu z języka `C++` na inne technologie.

W celu łatwiejszego zaprezentowania metod integracji języków `C++` i `rust` przygotowano specjalną bibliotekę[12]. Prezentuje ona sposób portowania kodu z jednej technologii na drugą.

7.4 Pierwsze próby implementacji metody `SALTGA`

W celu zaprezentowania działania portowania bibliotek języka `C++` do języka `rust` pierwotnie zakładano implementację metody w `C++`. Podjęte próby wdrożenia metody zakończyły się jednak niepowodzeniem spowodowanym problemami z zarządzaniem pamięcią w ramach programu. Inną przeszkodą, która uniemożliwiła zrealizowanie pierwotnego planu były ograniczone zasoby czasowe. Jak się okazało, portowanie bibliotek jest czasochłonnym zajęciem.

7.5 Napotkane problemy implementacyjne

- Zrównoleglanie kodu w `C++`, przy pomocy biblioteki `stpl`[13] okazało się czasochłonne i problematyczne. Przy ustawieniach optymalizujących budowę projektu w języku `C++` do statycznej biblioteki, `stpl` powodowało błędy kompilacji.
- Wiele roboczogodzin zostało skonsumowanych przez prosty błąd logiczny, związany z współdzieleniem pamięci między wątkami.
- Przy próbie zrównoleglenia, program powodował błąd pamięci[14]. Nie znaleziono powodu, ani rozwiązania dla tego problemu
- Portowanie kodu z języka `C++` na język programowania `rust` okazało się operacją, która wymagałaby znaczących zmian w kodzie.
- Gotowe biblioteki obsługujące sieci neuronowe, często są jednynie portami bibliotek z innych języków. To powoduje, że liczba funkcjonalności jest znacząco ograniczona. W wypadku omawianej pracy, okazało się, że wykorzystanie takich bibliotek nie byłoby wskazane, ponieważ nie udostępniały one dostępu do zmiennych (`varstore`). To wymusiło decyzję o napisaniu własnej implementacji sieci neuronowej.

[12] https://gitlab.dzordzu.pl/university/saltga_rust
[13] https://github.com/vit-vit/CTPL/blob/master/ctpl.h
[14] https://en.wikipedia.org/wiki/Segmentation_fault

- Korzystanie z plików binarnych czasem nie jest możliwe na starszych systemach operacyjnych. Rozwiązanie tego problemu zostanie poruszone w **Podrozdziale 8.3**

7.6 Wykonana implementacja metody `SALTGA`

Powyższe problemy spowodowały zmianę koncepcji rozwiązania. Ostatecznie główną część metody `SALTGA` zaimplementowano w ramach komponentu `backend` opisywanej pracy. Jedynie element dekompozycji został zaimplementowany jako oddzielna biblioteka napisana w języku `C++`.

7.7 Infrastruktura oprogramowania oraz CI/CD

Przygotowana infrastruktura zawierała poniższe elementy:

1. Własnoręcznie napisane skrypty wersjonujące. Uniwersalność skryptów jest zapewniona przez środowisko GITa.
2. Konfigurację `nginx` jako `reverse proxy`
3. Konfigurację `dockera`
4. Konfigurację serwisów `gitlab` oraz `nexus`
5. Konfigurację skryptów budujących w ramach systemu CI/CD
6. Konfigurację strefy domenowej[15]
7. Konfiguracji narzędzi automatyzujących budowę dokumentacji (`doxygen` oraz komenda `cargo docs`)

Niestety utrzymywanie infrastruktury było czasochłonne i powyższe elementy działały nieprzerwanie do początku grudnia 2021 roku. Po tym momencie maszyny budujące musiały zostać wyłączone. Obecnie pod domeną `drimt.im` znajduje się jedna ze starszych wersji aplikacji

8 Portowanie bibliotek z C++ na rusta

8.1 Wymagania wstępne

W celu połączenia biblioteki `decomposition` z komponentem `backend` wymagane było skorzystanie z bibliotek `cxx_build` oraz `cxx` dostępnych w ramach ekosystemu `cargo`. Dodatkowo element napisany w `C++` powinien być skompilowany statycznie, gdyż działanie na dynamicznych bibliotekach powodowało duże problemy z kompatybilnością pliku wykonywalnego. Dodatkowo wersjonowanie dynamicznych bibliotek było niezwykle uciążliwe. W momencie zainstalowania wielu wersji tego samego programu linker `C++` miał problemy ze znalezieniem niektórych plików.

8.2 Ogólny schemat działania

1. Należy zainstalować systemowo wymagane biblioteki, albo dodać ich pliki do katalogu z kodem aplikacji. W przypadku systemu `linux`, będą to te z rozszerzeniem `.a`
2. Kolejnym krokiem jest skonfigurowanie skryptu `build.rs`. Poniżej zamieszczono przykładową konfigurację skryptu budującego
3. Należy przygotować plik z kodem w `C++` tłumaczący kod biblioteki na kod zrozumiały dla biblioteki `cxx`. Dla poniższego przykładu będzie to plik `src/individual.cpp`

[15] drimt.im

4. Przygotować plik napisany w `rust`, który będzie wykorzystywał metody i klasy zdefiniowane w `src/individual.cpp`,
5. Gotowy kod portujący wykorzystać w swojej aplikacji

```
fn main() {
    let mut builder = cxx_build::bridge("src/individual.rs");
    builder
        .file("src/individual.cpp")
        .flag_if_supported("-std=c++17")
        .compile("salt-ga-rust");

    println!("cargo:rustc-link-search=/usr/local/lib/Dzordzu/");
    println!("cargo:rustc-link-lib=Decomposition");
    println!("cargo:rerun-if-changed=src/individual.cpp");
    println!("cargo:rerun-if-changed=src/individual.rs");
    println!("cargo:rerun-if-changed=include/individual.hpp");

}
```

8.3 Rozwiązywanie problemów z glibc

`glibc` jest to zestaw standardowych bibliotek wymaganych przez większość programów w systemie `linux`. Wielokrotnie zdarza się, że klastry obliczeniowe korzystają z starszych wersji systemu operacyjnego. W takim wypadku, o ile nie korzysta się z rozwiązań typu `musl`[16], należy wykorzystać program `patchelf`[17]. Modyfikuje on w pliku wykonywalnym ścieżki do kluczowych bibliotek.

9 Podsumowanie

9.1 Perspektywy rozwoju aplikacji

9.1.1 Rozwój metody `SALTGA`

Ograniczone zasoby czasowe, jak i problemy związane z implementacją rozwiązania uniemożliwiły przeprowadzenie badań nad jakością i wydajnością metody dla problemu optymalizacji sieci neuronowej. Dlatego przeprowadzenie badań i poprawa wydajności należy obecnie do perspektyw rozwoju aplikacji.

9.1.2 Rozdzielenie programu `backend`

Rozdzielenie programu umożliwiłoby łatwiejsze utrzymywanie kodu, a także zwiększyło atrakcyjność projektu dla programistów spoza zespołu inżynierskiego. Jest to kierunek na który aplikacja powinna zostać skierowana.

[16] https://www.musl-libc.org/faq.html
[17] https://github.com/NixOS/patchelf

9.1.3 Nowe metody optymalizacji

W przyszłości aplikacja mogłaby uzyskać dodatkowe rodzaje optymalizatorów. Ich zwiększona ilość znacząco poszerzyła by perspektywę na rozważany problem. Dodatkowo taki zabieg, mógłby zaowocować kolejnymi badaniami, a nawet nowymi odkryciami

9.1.4 Umożliwienie działania symulatora bez konieczności uruchamiania systemu webowego

Choć jest to zagadnienie ściśle związane z tym poruszonym w **Podrozdziale 9.1.2**, to warto zaznaczyć, że takie rozwiązanie uatrakcyjniłoby badaczom przygotowany system. W momencie, w którym konfiguracja symulatora odbywałaby się poprzez przekazywanie pliku w znanym formacie (np. `json`, czy `yaml`), odczucia użytkownika (*UX*) na pewno by się poprawiły.

9.1.5 Rozwój CI/CD

Ulepszona automatyzacja i konteneryzacja w dużym stopniu poprawiłaby dostępność oprogramowania. Jest to kierunek w który powinna zmierzać aplikacja.

9.1.6 Zastosowanie wytrenowanego modelu na realnym robocie minisumo

Jest to główna perspektywa rozwoju systemu. Praktyczne zastosowanie wytworzonego oprogramowania byłoby uwieńczeniem wszystkich trudów związanych z jego przygotowaniem. Choć autorzy byli świadomi ograniczeń czasowych, i nie wliczali tej funkcjonalności w obszar wymagań systemu, to jest to na pewno cel do realizacji w przyszłości.

9.2 Ocena projektu rozwiązania

Projekt rozwiązania (w ogólności) był dobrze przygotowany - jasna struktura, dobry wybór technologii, rozwinięta dokumentacja projektowa. Do błędów zaliczyć można jedynie początkowy plan implementacji całego algorytmu `SALTGA` w języku `C++`. Takie podejście przysporzyło wielu problemów i skutkowało stratą kilkuset roboczogodzin.

9.3 Ocena implementacji rozwiązania

Duża część kodu, jeśli nie całość jest napisana starannie. Redundancja kodu jest znikoma, a wykorzystane narzędzia zawsze mają swoje miejsce w projekcie - ich użycie jest zawsze uzasadnione.

Zakończenie

Choć pierwotnie zakładane cele nie zostały w pełni zrealizowane, bo zabrakło elementu badawczego, to udało się zaimplementować główny program. Dodatkowo sukcesem zakończyły się próby pokazania sposobu łączenia kilku technologii, oraz przygotowanie działającego CI/CD. Postawienie klarownych wymagań, rozważenie wielu dostępnych technologii i selekcja najlepszych (poza jednym, popełnionym błędem), wpłynęły na znakomitą jakość projektu. Co ciekawe, ten pojedynczy, aczkolwiek krytyczny błąd w doborze technologicznym nie zaprzepaścił przyszłości projektu. W ramach opracowywanego rozwiązania autorzy oprogramowania musieli opanować w stopniu przynajmniej dobrym elementy komunikacji wewnątrz projektu. Dodatkowo, wymiana doświadczeń pomiędzy członkami zespołu znacząco poprawiła ich znajomość rynku technologicznego i poszerzyła horyzonty. Pomimo swojej kompleksowości, wytworzone oprogramowanie wciąż posiada szeroką gamę możliwości rozwoju. Taki stan rzeczy umożliwia autorom rozwijanie swoich zainteresowań nawet po zakończeniu przedsięwzięcia inżynierskiego. W przyszłości wciąż istnieje możliwość na zrealizowanie wymarzonego celu - skutecznej implementacji wytrenowanej sieci w fizycznym robocie minisumo.

Spis tabel

Spis rysunków

Spis przypisów

Spis pseudokodów