



**POLITECHNIKA  
GDAŃSKA**

WYDZIAŁ ELEKTRONIKI,  
TELEKOMUNIKACJI I INFORMATYKI



Imię i nazwisko studenta: Oskar Piechowski

Nr albumu: 145526

Studia drugiego stopnia

Forma studiów: stacjonarne

Kierunek studiów: Automatyka i Robotyka

Specjalność/profil: -

## PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Optymalizacja struktury konwolucyjnych sieci neuronowych za pomocą algorytmu genetycznego

Tytuł pracy w języku angielskim: Optimization of convolutional neural networks structure using genetic algorithms

Potwierdzenie przyjęcia pracy	
Opiekun pracy	Kierownik Katedry/Zakładu (pozostawić właściwe)
podpis	podpis
dr inż. Mariusz Domżalski	

Data oddania pracy do dziekanatu:





## OŚWIADCZENIE

Imię i nazwisko: Oskar Piechowski

Data i miejsce urodzenia: 21.04.1993, Kościerzyna

Nr albumu: 145526

Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki

Kierunek: automatyka i robotyka

Poziom studiów: drugi

Forma studiów: stacjonarne

Ja, niżej podpisany(a), wyrażam zgodę/nie wyrażam zgody\* na korzystanie z mojej pracy dyplomowej zatytułowanej: Optymalizacja struktury konwolucyjnych sieci neuronowych za pomocą algorytmu genetycznego do celów naukowych lub dydaktycznych.<sup>1</sup>

Gdańsk, dnia ..... .....  
.....  
podpis studenta

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 2016 r., poz. 666 z późn. zm.) i konsekwencji dyscyplinarnych określonych w ustawie Prawo o szkolnictwie wyższym (Dz. U. z 2012 r., poz. 572 z późn. zm.),<sup>2</sup> a także odpowiedzialności cywilno-prawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza(y) praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

Potwierdzam zgodność niniejszej wersji pracy dyplomowej z załączoną wersją elektroniczną.

Gdańsk, dnia ..... .....  
.....  
podpis studenta

Upoważniam Politechnikę Gdańską do umieszczenia ww. pracy dyplomowej w wersji elektronicznej w otwartym, cyfrowym repozytorium instytucjonalnym Politechniki Gdańskiej oraz poddawania jej procesom weryfikacji i ochrony przed przywłaszczeniem jej autorstwa.

Gdańsk, dnia ..... .....  
.....  
podpis studenta

\*) niepotrzebne skreślić

<sup>1</sup> Zarządzenie Rektora Politechniki Gdańskiej nr 34/2009 z 9 listopada 2009 r., załącznik nr 8 do instrukcji archiwalnej PG.

<sup>2</sup> Ustawa z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym:

Art. 214 ustęp 4. W razie podejrzenia popełnienia przez studenta czynu podlegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.

Art. 214 ustęp 6. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 4, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o popełnieniu przestępstwa.



## **Streszczenie**

Streszczenie.

**Słowa kluczowe:** elektrownie wodne, systemy alarmowe, automatyzacja, sterowanie

**Dziedzina nauki i techniki, zgodnie z wymogami OECD:** nauki inżynierijne i techniczne, systemy automatyzacji i kontroli

## **Abstract**

English abstract.

**Keywords:** hydro plants, alarm systems, automation, control engineering

**Field of science and technology in accordance with OECD requirements:** engineering and technology, automation and control systems

## **Spis treści**

Wykaz ważniejszych oznaczeń i skrótów .....	6
1. Wstęp .....	8
2. Przegląd stanu wiedzy .....	10
2.1. Uczenie maszynowe .....	10
2.2. Metauczenie .....	10
2.3. Uczenie z nadzorem .....	11
2.4. Klasyfikacja obrazów .....	11
2.5. CIFAR-10 .....	12
2.6. K najbliższych sąsiadów .....	14
2.7. Klasyfikator liniowy .....	15
2.8. Sieci Neruonowe .....	17
2.9. Konwolucyjne sieci neuronowe .....	18
2.10. Algorytm genetyczny .....	20
3. Opis rozwiązania .....	23
3.1. Redukcja problemu .....	23
3.2. Sieć neuronowa jako osobnik algorytmu genetycznego .....	23
3.3. Operacje genetyczne algorytmu genetycznego .....	24
3.3.1. Selekcja .....	24
3.3.2. Krzyżowanie .....	25
3.3.3. Mutacja .....	25
3.3.4. "Naprawianie" osobników .....	25
3.3.5. Substytucja .....	26
3.4. Architektura eksperimentu .....	26
3.4.1. Elementy wspólne .....	27
3.4.2. Węzeł zarządzający .....	27
3.4.3. System kolejkowania zadań .....	29
3.4.4. Protokół komunikacji ze zdalnymi węzłami .....	30
3.4.5. Węzeł obliczeniowy .....	31
3.4.6. Zarządzanie infrastrukturą węzłów obliczeniowych .....	33
3.5. Replikatywność badań .....	37
4. Badania i testy .....	38
4.1. Wariancja wyników dla jednej sieci .....	38
4.2. Test działania algorytmu genetycznego .....	38
4.3. Test działania dla lokalnego węzła obliczeniowego .....	41
4.4. Poszukiwanie optymalnej struktury konwolucyjnej sieci neuronowej za pomocą algorytmu genetycznego dla uczenia 1-epokowego .....	42

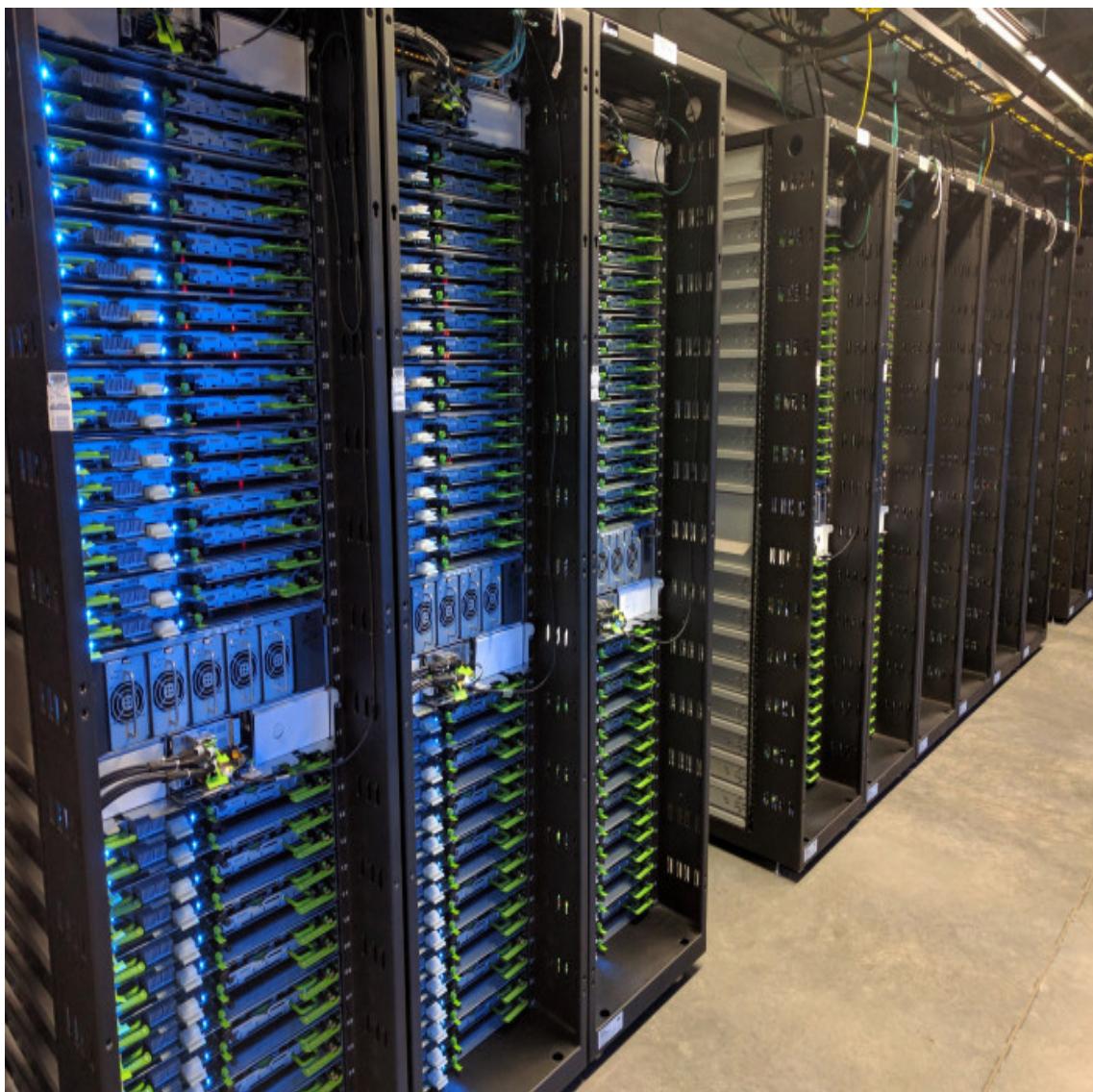
4.5. Porównanie wyników dla uczenia 1- i 10- epokowego .....	45
4.6. Porównanie przebiegu uczenia sieci w pierwszej i ostatniej iteracji algorytmu genetycznego .....	46
4.7. Osiągi najlepszej według algorytmu sieci po 25 epokach uczenia .....	47
5. Podsumowanie .....	49
Bibliografia .....	51
Spis rysunków .....	52
Spis tabel .....	53
Dodatek A .....	54
Dodatek B .....	56
Dodatek C .....	70
Dodatek D .....	71

## **Wykaz ważniejszych oznaczeń i skrótów**

- $W$  – Macierz z wagami połączeń sieci neuronowej  
 $x_i$  – Piksele obrazka wejściowego spłaszczone do jednowymiarowego wektora  
KSN – Konwolucyjna Sieć Neronowa  
AG – Algorytm Genetyczny  
SI – Sztuczna Inteligencja

## Wstęp

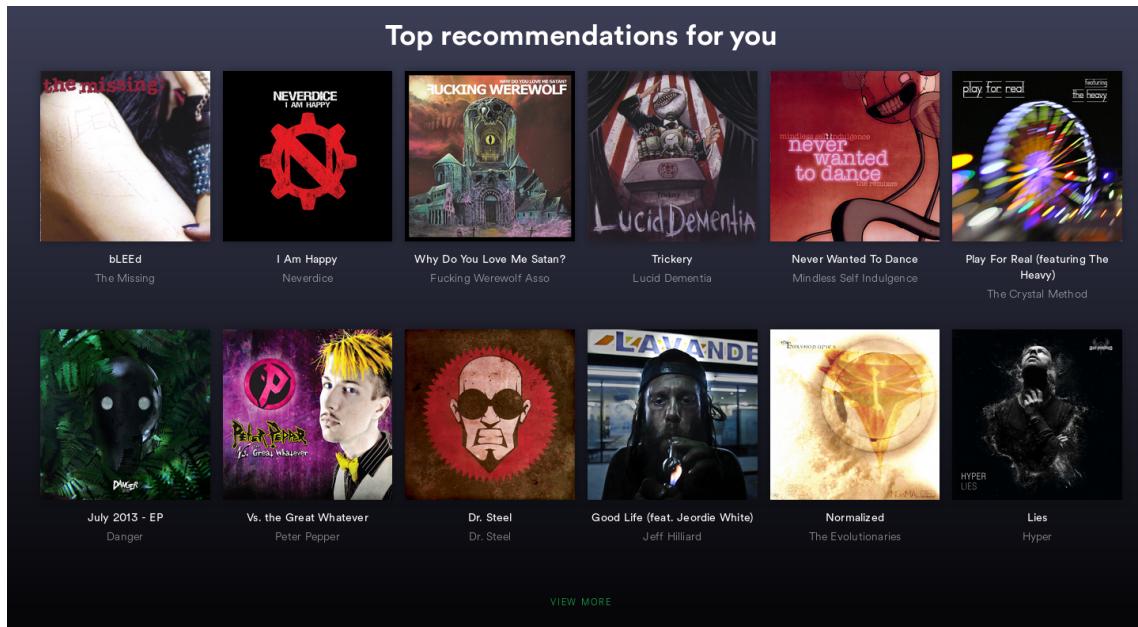
W dobie powszechnego dostępu do internetu i informatyzacji człowiek człowiek produkuje olbrzymie ilości danych. Każdego dnia setki milionów ludzi robią zdjęcia, filmy i wysyła wiadomości tekstowe. Rządy państw regularnie zbierają dane wszelkiego rodzaju, od spisów statycznych po raporty incydentów od jednostek policji. Ogrom tych danych rośnie w dużym tempie. Całkowita liczba danych na świecie wyniosła 4,4 zettabajów ( $4,4 \times 10^{21}$ ) danych. Prognozuje się dalszy wzrost do 44 zettabajtów do roku 2020. Szacuje się, że dziennie powstaje ok. 2,5 exabajtów ( $2,5 \times 10^{18}$  bajtów) danych. [9] Z jednej strony jest to wyzwanie, gdyż aby przetwarzać tak wielką liczbę danych potrzeba wielu zasobów - sieciowych, obliczeniowych i magazynowych. Jedno z centrum obliczeniowych (*Data Center*) firmy *Facebook* w miejscowości *Prineville* w stanie *Oregon* składa się z 4 budynków o łącznej powierzchni ok.  $42000m^2$ . Poniżej, na rys. 1.1 przedstawiono 8 szaf serwerowych (*rack*), każda złożona z 32 serwerów i mogącą przechować 2 petabajty danych. [11]. Na  $42 \text{ tys. } m^2$  można rozmieścić wiele takich szaf, co w wyniku daje olbrzymie możliwości, a to zaledwie jeden z wielu takich kompleksów na świecie.



Rys. 1.1. Fragment centrum obliczeniowego  
Źródło: [11]

Duża liczba danych niesie ze sobą zarówno wyzwania, jak i korzyści. W ostatnich latach ukuło się w języku angielskim pojęcie *Big Data*, które definiujemy jako duże zbiory danych zarówno ze zbiorów tradycyjnych jak i cyfrowych, które są źródłem dla nowych odkryć i analiz. [2]. Przetwarzanie

i analiza owych zbiorów danych jest skomplikowana i czasochłonna dla człowieka, dlatego z pomocą przychodzą algorytmy z dziedziny sztucznej inteligencji, a konkretnie ich podzbior związany z uczeniem maszynowym *Machine learning*, zdefiniowany dalej w rozdziale 2. Typowymi zastosowaniem tego typu algorytmów jest filtr antyspamowy, które klasyfikuje wiadomości jako spam na podstawie dużego zbioru wiadomości oznaczonych jako spam lub nie. Innym popularnym zastosowaniem są systemy rekommendacji - za przykład niech posłuży system w odtwarzaczu *Spotify*, który na podstawie słuchanej przez użytkownika muzyki dobiera dla niego propozycje, które również mogą mu się spobodać, ze względu na podobieństwo. Przykładowe rekommendacje przedstawiono na rys. 1.2.



Rys. 1.2. Rekomendowana przez algorytmy uczenia maszynowego muzyka w serwisie *Spotify*  
 Źródło: praca własna

Rozwiązań tego typu stosuje wiele innych popularnych serwisów multimedialnych, między innymi *YouTube*, *Netflix*. Algorytmy tego typu, co zostanie przybliżone w rozdziale 2, uczą się na podstawie podanych im danych, każdy jednak taki algorytm (np. sieci neuronowe), opisać można przez kilka parametrów. Parametry owe można dobierać w sposób arbitralny, można jednak również do wyznaczania ich optymalnych nastaw zaprzecząc innemu algorytmowi (należący również klasy algorytmów SI lub nie), np. algorytm genetyczny, algorytmy roju itd.. Inspiracją do przeprowadzenia badań w tym kierunku były zajęcia laboratoryjne z przedmiotu Sztuczna Inteligencja w Automatyce na studiach inżynierskich na kierunku Automatyka i Robotyka na wydziale Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej. Jedno z zadań laboratoryjnych polegało na znalezieniu optymalnych parametrów sieci neuronowej, tak aby wynikowa sieć dobrze aproksymowała zadaną funkcję. Korzystając z pewnej wiedzy na temat samych sieci neuronowych oraz dobierania liczby neuronów, rodzaju funkcji aktywacji, liczby warstw ostatecznie metodą prób i błędów dochodziło się do zadowalającego rozwiązania. W tym miejscu pojawiła się idea, aby nie robić tego ręcznie, a użyć innego algorytmu. Stąd wysunąć można główną tezę tej pracy - że możliwa jest optymalizacja owych parametrów za pomocą algorytmu genetycznego. Dodatkowo, w związku z tym, że pełne uczenie większych sieci dla bardziej skomplikowanych problemów jak klasyfikacja obrazów jest czasochłonne, poszukiwano sposobu na skrócenie owego czasu.

Poprzez analogię do świata ludzi, gdzie można dla wielu przypadków zaobserwować prawidłowość, że mądre dzieci szybko się uczą i ostatecznie dochodzą do lepszych wyników jako dorosły, niż dzieci mniej rozgarnięte, wysnuto dodatkową tezę, mianowicie, że istnieje związek pomiędzy wynikami danej sieci neuronowej po uczeniu przez jedną epokę a jej finalną skutcznością po uczeniu pełnym. Dalej przedstawiony zostanie przegląd stanu wiedzy w rozdziale 2, zostanie opisany system, który powstał w celu udowodnienia owych tez w rozdziale 3 oraz omówione zostaną wyniki badań w rozdziale 4. Wszystko zostanie podsumowane w rozdziale 5.

## Przegląd stanu wiedzy

### ***Uczenie maszynowe***

Mówiąc, że program komputerowy uczy się z doświadczenia (*experience*)  $E$  w związku z pewną klasą zadań (*tasks*)  $T$  oraz miarą wydajności (*performance measure*)  $P$  jeżeli jego osiągi w zadaniach w  $T$ , jak mierzone przez  $P$ , ulegają polepszeniu z doświadczeniem  $E$ . [14] W przypadku pojedynczej konwolucyjnej sieci neuronowej (opisanej bliżej w 2.9):

- jako doświadczenie  $E_{CN}$  możemy zdefiniować przetwarzanie obrazów wraz z ich etykietami klas
- zadaniem  $T_{CN}$  nazwiemy klasyfikację obrazów
- miarą jakości  $P_{CN}$  będzie stosunek liczby poprawnie zaklasyfikowanych zdjęć ze zbioru testowego do wielkości zbioru testowego

W przypadku algorytmu genetycznego (opisanego w 2.10) poszukującego optymalnej struktury sieci neuronowej:

- doświadczeniem  $E_{GA}$  będzie tworzenie i testowanie nowych sieci neuronowych
- zadanie  $T_{GA}$  to poszukiwanie optymalnej w sensie  $P_{CN}$  struktury (parametrów) sieci neuronowej
- miarą jakości  $P_{GA}$  będzie miara jakości najlepszego wygenerowanego przez algorytm osobnika

Zastosowanie jednego algorytmu z dziedziny uczenia maszynowego do poprawy procesu uczenia się innego algorytmu tej samej klasy ma ścisły związek z pojęciem metauczenia.

### ***Metauczenie***

1. System metauczający się (*A metalearning system*) musi zawierać podsystem uczący się, który przystosowuje się z doświadczeniem.
2. Doświadczenie jest zdobywane poprzez wykorzystywanie metawiedzy wydobytnej
  - a) z poprzedniego epizodu uczenia na pojedynczym zbiorze danych, i/lub
  - b) z innych dziedzin lub problemów.

Dodatkowo, często używanym pojęciem w metauczeniu jest pojęcie tendencjonalności (*bias*), które w tym kontekście odnosi się do zbioru założeń wpływających na wybór hipotez opisujących dane. [12] Hipoteza w tym kontekście oznacza funkcję predykcyjną (*predictive function*) powstającą w wyniku zastosowania tradycyjnego uczenia (np. konwolucyjnej sieci neuronowej) pewnymi danymi, a wpływ na jej kształt mają ustalone założenia zaszyte w strukturze uczonego algorytmu. [6, s.2] Autorzy w [6] wyróżniają:

- tendencjonalność deklaratywną (*declarative bias*) - specyfikuje ona reprezentację przestrzeni hipotez, np. przedstawianie hipotez korzystając wyłącznie z sieci neuronowych
- tendencjonalność proceduralną (*procedural bias*) - wpływa na szeregowanie hipotez, np. preferowanie hipotez o krótszym czasie wykonania (*runtime*)

Zgodnie z tą teorią, w tradycyjnym uczeniu tendencjonalność jest stała, podczas gdy metauczenie stara się ją dobierać dynamicznie. [12] W kontekście projektowanego systemu automatycznie dobierającego parametry konwolucyjnej sieci neuronowej, spełnia ona założenia sprecyzowane w pierwszej części definicji, tj.

- Istnieje uczący się podsystem (konwolucyjna sieć neuronowa)
- Struktura najlepszego osobnika ulega zmianie (przystosowuje się z doświadczeniem)
- Metawiedza jest wydobywana z poprzednich epizodów uczenia (jako wartość funkcji przystosowania konkretnej struktury)

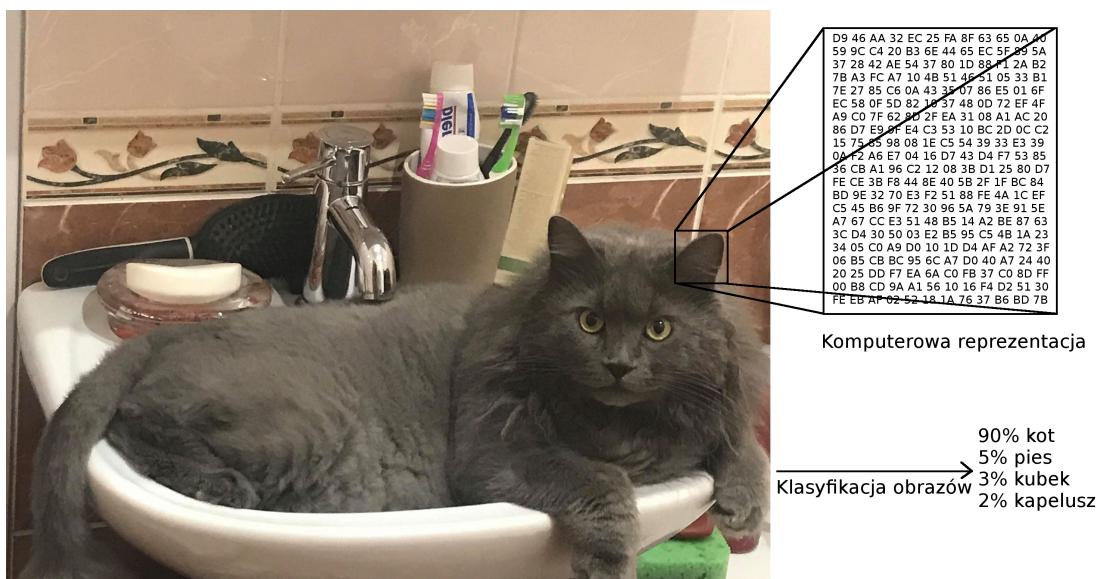
W odniesieniu do tendencyjności deklaratywnej - przestrzeń poszukiwań (najlepszego algorytmu do klasyfikacji obrazów) została zawężona wyłącznie do konwolucyjnych sieci neuronowych, jednakże sieci o różnych strukturach będą produkować różne hipotezy. Poszukiwanie najlepszego algorytmu do klasyfikacji obrazów na zbiorze CIFAR-10 nie jest tematem tej pracy - jest nią próba znalezienia optymalnej struktury dla konkretnego algorytmu. W przypadku tak zawężonej przestrzeni poszukiwań dalej możemy preferować jedne generowane hipotezy od drugich, co można regulować odpowiednio modyfikując funkcję przystosowania, np. uwzględniając karę za komplikację uzyskanej struktury (np. wynikowa ilość aktywnych warstw konwolucyjnych).

### **Uczenie z nadzorem**

Zadanie uczenia maszynowego nauczenia się funkcji wiążącej wejście z wyjściem na podstawie przykładowych par wejściowo-wyjściowych nazywamy uczeniem z nadzorem. [16] Polega ono na wywołaniu funkcji z oznaczonych etykietami danych treningowych składających się ze zbioru przykładów treningowych. [15] Przykładowy problem, który może zostać zaklasyfikowany do uczenia z nadzorem: posiadając historyczne dane miesięczne na temat ilości pożarów w danym mieście, przewidzieć ile pożarów wybucha w nadchodzącym miesiącu. W tym wypadku czas (miesiąc) jest zmienną niezależną, a liczba pożarów zmienną zależną. Ustalenie związku pomiędzy tymi wielkościami (znalezienie modelu) jest zadaniem regresji, które jest jedną z podklas uczenia z nadzorem. Drugą klasą takich problemów jest tzw. problem klasyfikacji, w którym, zamiast wyznaczać funkcję, której zbiór wartości jest (w przybliżeniu) ciągły, identyfikujemy odwzorowane, które pozwalą nam nową, nieznaną daną zakwalifikować do określonego zbioru klas. Popularnym przykładem jest uczenie systemu klasyfikowania wiadomości e-mail jako spam lub nie, gdy posiadamy zbiór treningowy w postaci już opisanych jako należące do odpowiedniej klasy wiadomości. Podczas gdy taki klasyfikator (filtr antyspamowy) przetwarza tekst, w przypadku niniejszej pracy magisterskiej mamy do czynienia z problemem klasyfikacji obrazów.

### **Klasyfikacja obrazów**

Klasyfikacja obrazów jest zadaniem z klasy uczenia z nadzorem, w którym poszukujemy funkcji przypisującej obrazom odpowiednie etykiety z pewnego skończonego zbioru. Jest to jeden z podstawowych problemów przetwarzania obrazów. Co więcej, wiele pozornie różnych zadań (takich jak detekcja obiektów, segmentacja) można zredukować do problemu klasyfikacji obrazów. Przykładowo, model klasyfikujący obrazy przetwarza zdjęcie pokazane poniżej na rys. 2.1 i przypisuje prawdopodobieństwa do czterech kategorii, *{kot, pies, kapelusz, kubek}*. Jak pokazano na rysunku, należy pamiętać, że zdjęcia dla komputera są niczym innym jak dużą trójwymiarową tablicą liczb. W tym przypadku zdjęcie kota jest szerokie na 2048 pikseli, wysokie na 1536 pikseli i posiada 3 kanały koloru: czerwony, zielony i niebieski (z ang. w skrócie *RGB*). Tym samym zdjęcie składa się z  $2048 \times 1536 \times 3 = 9437184$  liczb.



Rys. 2.1. Komputerowa reprezentacja zdjęcia kota  
Źródło: praca własna na podstawie [13]

Zadanie klasyfikacji obrazów w tym przypadku polega na zredukowaniu tych prawie 10 milionów liczb do jednej etykiety, dla rys. 2.1 będzie to "kot". Dla człowieka rozpoznanie obiektu jest względnie proste. Z punktu widzenia algorytmu przetwarzania obrazów napotykamy kilka zasadniczych trudności, związanych z faktem, że komputer interpretuje obraz jako trójwymiarową macierz wartości jasności pikseli:

- Zróżnicowanie perspektywy - ten sam obiekt może być różnie zorientowany względem kamery rejestrującej obraz
- Zróżnicowanie skali - obiekty jednej klasy (np. koty) często występują w różnych rozmiarach (nie tylko z punktu widzenia rozmiaru na zdjęciu, ale i w rzeczywistości)
- Deformacja - wiele klasyfikowanych obiektów nie jest bryłami sztywnymi i mogą być skrajnie zniekształcone
- Okluzja - zjawisko zasłonięcia obiektu przez inny obiekt, czasem jedynie niewielki fragment szukanego obiektu jest widoczny
- Warunki oświetleniowe - efekty różnego oświetlenia są bardzo wydatne na poziomie pikseli
- Nieład w tle - klasyfikowane obiekty mogą zlewać się z tłem, czyniąc ich identyfikację trudniejszą
- Zróżnicowanie wewnętrz klasy - niektóre klasy są bardzo ogólne (np. broń) i zawierają w sobie bardzo różnie wyglądające obiekty (np. miecz a karabin maszynowy).

Na rys.2.2 przedstawiono ilustrację powyższych problemów.



Rys. 2.2. Problemy w klasyfikacji obrazów  
 Źródło: praca własna na podstawie [13]

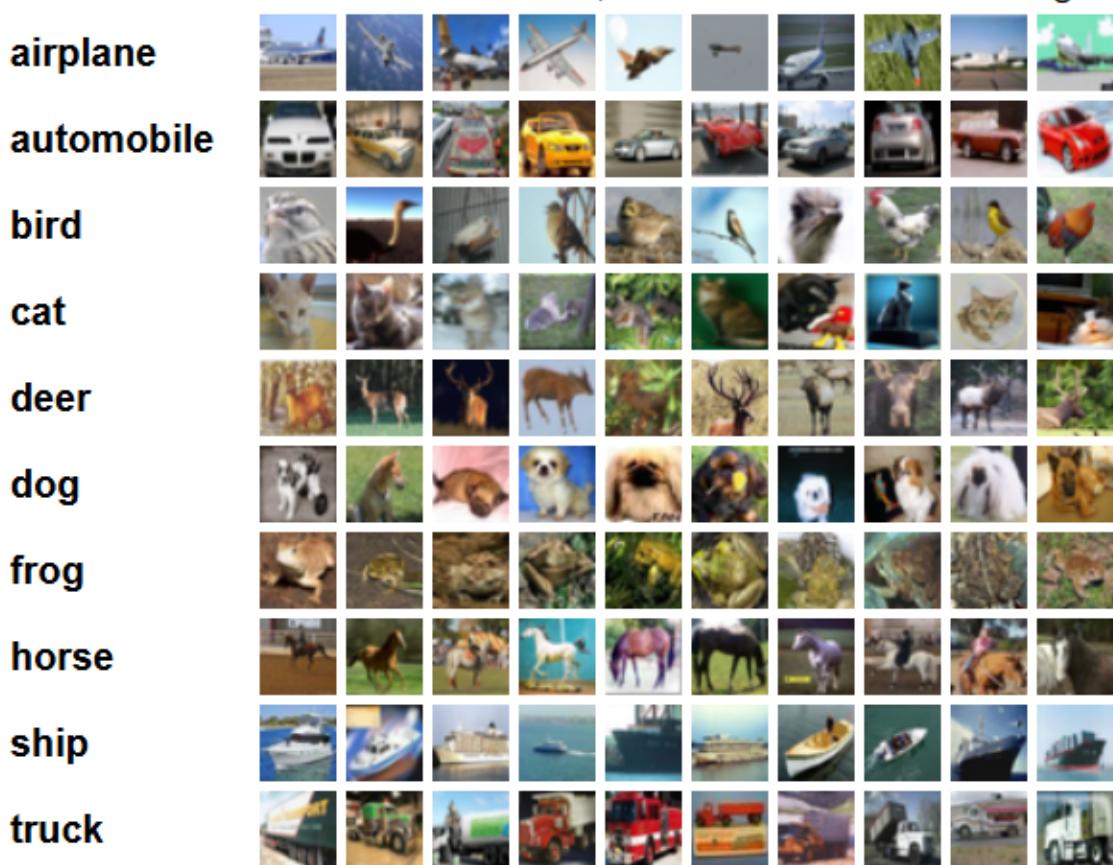
Dobry model klasyfikujący obrazy powinien być niewrażliwy na dowolne kombinacje powyższych zakłóceń, jednocześnie pozostając wrażliwym na zróżnicowanie pomiędzy klasami. Stworzenie algorytmu poprawnie klasyfikującego obrazy jest zadaniem nietrywialnym (w przeciwieństwie np. do ułożenia algorytmu sortującego liczby według zadanego porządku). Jednym ze podejść, jakie można przyjąć do tego problemu, jest stworzenie algorytmu, który będzie się uczył na podstawie dużej ilości oznaczonych etykietami danych, czyli wspomniane wcześniej uczenie maszynowe (2.1) i uczenie z nadzorem (2.3). Takie rozwiązanie problemu nazywane jest również podejściem opartym na danych (*data-driven approach*). [13]

### CIFAR-10

Przykładem zbioru danych służącego do uczenia i oceny algorytmów uczenia maszynowego w dziedzinie klasyfikacji obrazów jest zbiór CIFAR-10. Zbiór danych CIFAR-10 składa się z 60000 kolorowych obrazków o wymiarach 32x32 pogrupowanych w 10 klas, po 6000 obrazków w każdej. Zbiór trenigowy składa się z 50000 obrazków, a testowy z 10000.

Dane zorganizowane są w pięć partii trenigowych i jedną testową, po 10000 obrazków każda. Partia testowa zawiera dokładnie po 1000 losowo wybranych obrazków z każdej klasy. Partie treningowe zawierają pozostałe obrazki w losowej kolejności, w związku z czym niektóre partie mogą zawierać więcej obrazków z jednej klasy niż pozostałych. Łącznie zawierają po 5000 obrazków z każdej klasy.

Poniżej, na rysunku 2.3 przedstawiono wszystkie klasy wraz z 10 losowymi obrazkami dla każdej z nich.



Rys. 2.3. Przykładowe obrazki ze zbioru CIFAR-10

Źródło: <https://www.cs.toronto.edu/~kriz/cifar.html>

Każda z klas wyklucza się wzajemnie. Klasy "samochód" (*automobile*) i "ciężarówka" (*truck*) nie posiadają części wspólnej. "Samochód" zawiera sedany, SUVy i inne tego typu samochody. "Ciężarówka" zaiwera tylko duże ciężarówki. Żadna z nich nie zawiera pojazdów typu pickup.

Poniżej opisano organizację tego zbioru danych w wersji dla języka Python.

Archiwum zawiera pliki `data_batch1`, `data_batch2` `data_batch5`, jak również `test_batch`. Każdy z tych plików jest poddany serializacji wersją obiektu Pythonowego wytworzzonego za pomocą `cPickle`. Poniżej, na listingu 2.1 przedstawiono procedurę odczytującą te pliki i zwracającą słownik dla Pythona w wersji 2:

```
def unpickle(file):
    import cPickle
    with open(file, 'rb') as fo:
        dict = cPickle.load(fo)
    return dict
```

Listing 2.1. Procedura ładowania zbioru danych

Załadowane w ten sposób, każdy z plików z partiami zawiera następujące elementy:

- *data* - macierz *numpy* typu *uint8* o wymiarach 10000x3072. Każdy rzęd macierzy przechowuje kolorowy obrazek o wymiarach 32x32. Pierwsze 1024 pozycji zawiera wartości pikseli dla kanału czerwononego, kolejne 1024 dla zielonego, a ostatnie dla niebieskiego. Zdjęcia przechowywane są rzędami w kolejności od góry do dołu, tj. pierwsze 32 elementy macierzy opisują wartości kanału czerwonego dla pierwszego rzędu pikseli obrazka.
- *labels* - lista 10000 liczb w przedziale 0-9. Liczba pod indeksem i oznacza etykietę z klasą dla i-tego obrazka w macierzy *data*.

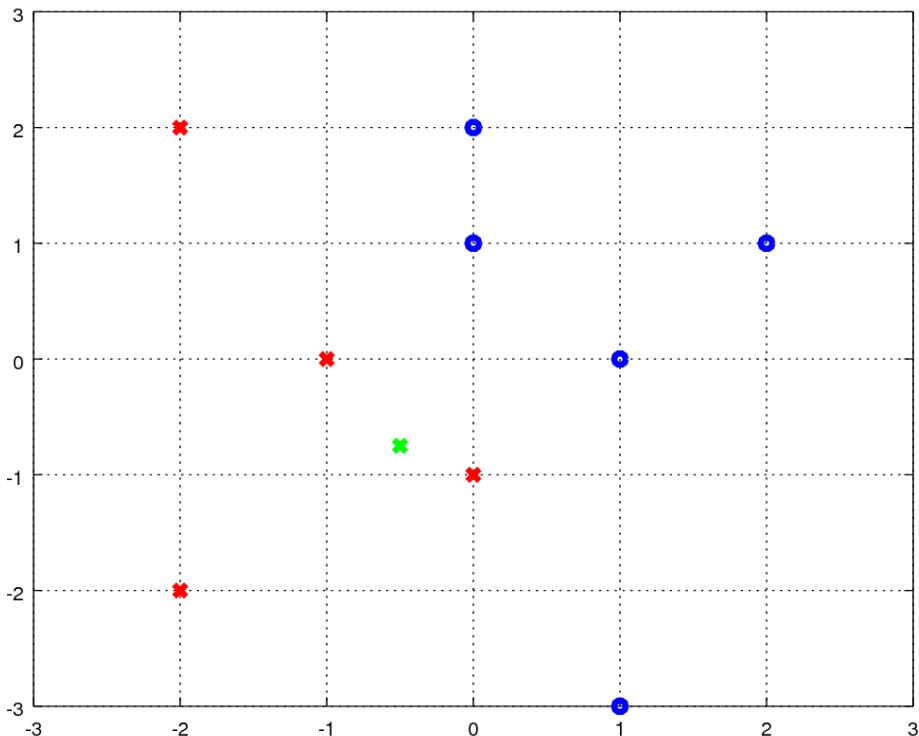
Zbiór danych zawiera również plik batches.meta, który również zawiera obiekt typu słownik dla języka Python. Zawiera on następujące elementy:

- *label\_names* - 10 elementowa lista która nadaje czytelne dla człowieka nazwy liczbowym etykietom w liście *labels* opisanej powyżej. Na przykład, *label\_names[0] == "airplane"*, *label\_names[1] == "automobile"*, itd. [10]

Wśród kilku podejść, które mogą nam posłużyć do rozwiązania tego problemu, wyróżnić możemy algorytm k najbliższych sąsiadów ( $k$  - *Nearest Neighbour Classifier*), klasyfikatory liniowe, sieci neuronowe.

### **K najbliższych sąsiadów**

Dla zbioru CIFAR-10 mamy 50000 obrazków oznaczonych etykietami i 10000 obrazów, które chcemy zaklasyfikować. Każdy obrazek o wymiarach  $32 \times 32 \times 3$  możemy przedstawić jako wektor o wymiarach  $3072 \times 1$ . Oznacza to, że każdy z nich możemy interpretować jako punkt w przestrzeni 3072-wymiarowej. Najproszym sposobem w jaki możemy klasyfikować nowe dane jest badanie ich położenia w owej przestrzeni. Zakładając, że obrazki jednej klasy będą grupować się w klastry, każdy kolejny punkt znajdujący się blisko chmury punktów reprezentującej daną klasę z wysokim prawdopodobieństwem również będzie przynależał do tej klasy. Przykładowo, rozważmy sytuację jak na rys. 2.4 dla klasyfikacji dwuwymiarowych punktów. Punkt albo przynależy do klasy 'niebieski okrąg', albo 'czerwony krzyżyk'. Rozważamy punkt, który oznaczony jest zielonym krzyżykiem.



Rys. 2.4. Ilustracja działania algorytmu k najbliższych sąsiadów w dwóch wymiarach  
 Źródło: praca własna

Dla najproszego przypadku  $k = 1$  badamy odległości (w tym przypadku używając normy  $l^2$ ) pomiędzy interesującym nas punktem a wszystkimi danymi ze zbioru treningowego i przyjmujemy, że badany punkt będzie przynależał do tej samej klasy, do której przynależy jej najbliższy sąsiad. W przypadku rys. 2.4 badany zielony punkt znajduje się najbliżej czerwonego krzyżka w punkcie  $(0, -1)$ , zatem przypisujemy go do tej samej klasy. Dla większych wartości  $k$  rozważamy więcej najbliższych sąsiadów i przypisujemy rozważany punkt do tej klasy, do której należy większość najbliższych punktów.

Podejście to, mimo iż intuicyjne i proste w implementacji, nie rozwiązuje dobrze problemu klasyfikacji, osiągając skuteczność 39.6% dla  $k = 1$  dla zbioru CIFAR-10. Jest to lepiej niż losowe przydzielanie do klas (które dla 10 klas ma skuteczność 10%), ale wciąż dalekie od skuteczności człowieka.

Jednym z problemów podejścia k najbliższych sąsiadów jest kosztowne testowanie nowych punktów. Dla każdego nowego punktu wymaga ono policzenia odległości od 50000 punktów w przestrzeni 3072-wymiarowej oraz posortowania wg. obliczonej odległości, co jest dość kosztowną operacją.

Kolejnym problemem w przypadku przetwarzania obrazów jest klasyfikowanie wyłącznie na podstawie dominujących w obrazie wartości kolorów. Na przykład, zarówno w przypadku klasy "samolot" jak i w przypadku klasy "statek" często występującym w tle kolorem będzie niebieski, gdyż niebieskie jest zarówno niebo i woda. Tak duże podobieństwo może prowadzić do błędnej klasyfikacji w przypadku tych dwóch klas. [13]

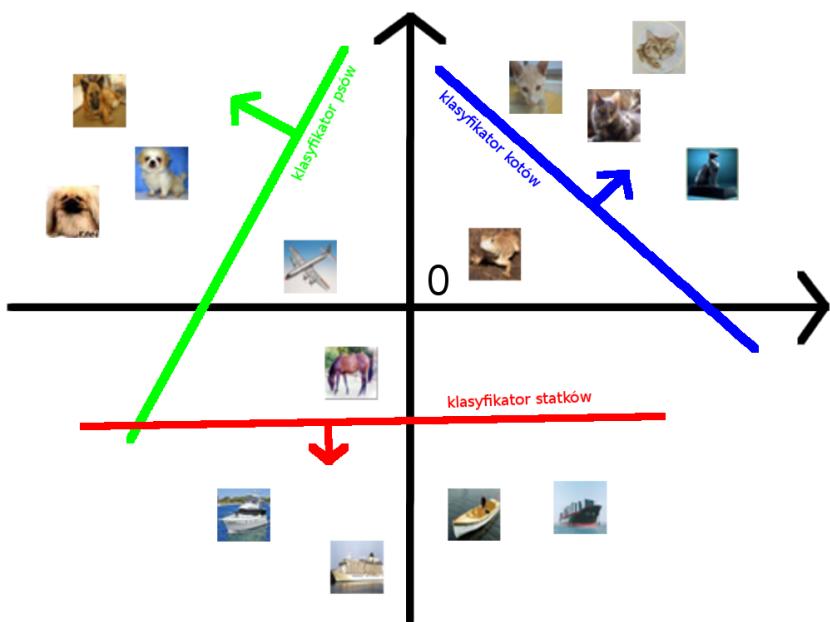
### Klasyfikator liniowy

Innym podejściem, jakie można przyjąć do klasyfikacji obrazów, jest stworzenie liniowego odwzorowania pomiędzy pikselową reprezentacją obrazu a stopniem przynależności danego obrazu do poszczególnych klas. Oznaczając dane treningowe przez  $x_i \in R^D$ , każde powiązane z etykietą  $y_i$ . W tym przypadku  $i = 1 \dots N$  i  $y_i \in 1 \dots K$ , tj. zbiór uczący składa się z  $N$  przykładów (każdy D-wymiarowy) i  $K$  rozróżnialnych kategorii. Przykładowo, dla zbioru CIFAR-10 zbiór treningowy ma rozmiar  $N = 50000$ , każdy po  $32 \times 32 \times 3 = 3072$  pikseli i  $K = 10$ , ponieważ rozróżniamy pomiędzy 10 klasami (pies, kot, samochód itd.). Zdefiniujemy funkcję oceny (*score function*)  $f : R^D \mapsto R^K$ , która odwzorowuje wartości pikseli do oceny przynależności do danej klasy. Zatem liniowy klasyfikator zdefiniować jako następujące liniowe odwzorowanie:

$$f(x_i, W, b) = Wx_i + b \quad (2.1)$$

We wzorze 2.1 zakładamy, że obrazek  $x_i$  zawiera wszystkie wartości swoich pikseli w wektorze o wymiarach  $[D \times 1]$ . Macierz  $W$  (o wymiarach  $[K \times D]$ ) oraz wektor  $b$  (o wymiarach  $[K \times 1]$ ) są parametrami funkcji. W zbiorze CIFAR-10  $x_i$  zawiera wszystkie piksele  $i$ -tego obrazka spłaszczone do jednej  $[3072 \times 1]$  kolumny,  $W$  ma wymiar  $[10 \times 3072]$  a  $b$   $[10 \times 1]$ , zatem 3072 liczby są wejściem dla funkcji, a wyjście jest 10 (oceny klas). Parametry wewnętrz  $W$  nazywane są również wagami, a wektor  $b$  nazywany jest wektorem przesunięcia (*bias vector*), ponieważ ma wpływ na wyjściowe oceny bez wchodzenia w interakcję z danymi  $x_i$ . [13]

Jak już wspomiano, obrazki mogą być rozumiane jako punkty w 3072 wymiarowej przestrzeni. Pojedynczy liniowy klasyfikator możemy interpretować jako hiperplaszczyznę separującą punkty w tej przestrzeni. Możemy wyobrazić sobie ściśnięcie wszystkich punktów do dwóch wymiarów jak na rys. 2.5, wtedy hiperplaszczyzny sprawdzają się do prostych.

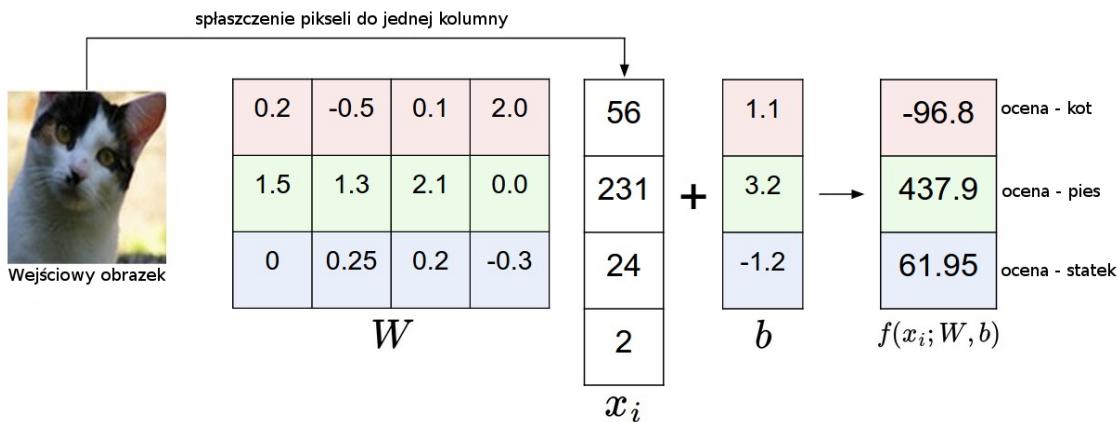


Rys. 2.5. Schematyczna wizualizacja działania klasyfikatorów liniowych  
Źródło: praca własna na podstawie [13]

Przy klasyfikatorze liniowym warto zauważać kilka faktów:

- Pojedyncze mnożenie macierzowe  $Wx_i$  w efekcie wyznacza oceny dla 10 różnych klasyfikatorów jednocześnie (jeden na klasę), gdzie każdy klasyfikator to rząd macierzy  $W$  i jednej prostej separującej na rys. 2.5
- Dane wejściowe  $(x_i, y_i)$  są ustalone, a kontrolujemy parametry  $W, b$ . Naszym celem będzie ustalenie tych parametrów w taki sposób, by wynikowe oceny zgadzały się z prawdziwymi przynależnościami do klas na całym zbiorze treningowym.
- Zaletą tego podejścia jest możliwość odrzucenia (nie przechowywania w pamięci) zbioru treningowego po nauczeniu się parametrów  $W, b$ . Dzieje się tak, ponieważ nowe punkty do klasyfikacji klasyfikujemy przy pomocy ocen uzyskanych z już nauczanej funkcji.
- Klasyfikowanie obrazów testowych sprowadza się do pojedynczego mnożenia macierzowego i dodawania, które jest szybsze od porównywania do wszystkich obrazków ze zbioru treningowego (jak w 2.6).

Przykład mapowania wartości pikseli do oceny przynależności do klas przedstawiono na rys. 2.6



Rys. 2.6. Mapowanie 4 monochromatycznych pikseli do 3 klas  
 Źródło: praca własna na podstawie [13]

Jak widać na rys. 2.6, wagi w macierzy  $W$  nie są dobrze dobrane, gdyż klasyfikator przypisuje najwyższą ocenę klasie pies, podczas gdy w istocie na obrazku znajduje się kot. Uczenie liniowego klasyfikatora będzie polegać na takim doborze wag, by błędy klasyfikacji zachodziły jak najrzadziej. W tym celu zdefinujemy funkcję kosztów (*loss function*), która będzie miarą tego jak dobrze działa dany klasyfikator.[13]

Funkcję kosztów dla całego zbioru danych definiuje wzór 2.2.

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{składowa danych}} + \underbrace{\lambda R(W)}_{\text{składowa regularizacji}} \quad (2.2)$$

gdzie  $\lambda$  jest parametrem kontrolującym oddziaływanie regularizacji przy uczeniu,  $L_i$  wartością funkcji kosztu dla pojedynczego obrazka przy ustalonych wagach  $W$  (wzór 2.4 lub 2.5). Składowa regularizacji penalizuje duże wagi w macierzy  $W$  i w tym przypadku będzie definiowana jako norma  $l^2$  dla tej macierzy, zgodnie ze wzorem 2.3.

$$R(W) = \sum_k \sum_l W_{k,l}^2 \quad (2.3)$$

Wprowadzenie tej składowej do funkcji kosztu pozwala uniknąć niejednoznaczności rozwiązania, gdzie dowolna macierz  $W$  przeskalowana o współczynnik  $\alpha \in R^+$  jest równie dobrym rozwiązaniem. W ten sposób prefeorowane będą rozwiązania o mniejszych wartościach współczynników. Co więcej, okazuje się, że przy takim podejściu uzyskujemy lepiej generalizujące i bardziej odporne na nadmierne dopasowanie (*overfitting*) klasyfikatory. Wynika to z faktu, że żaden z wejściowych wymiarów nie może

mieć dużego wpływu na końcowy wynik. Dla przykładu, dla wektora danych  $x = [1, 1, 1, 1]$  i dwóch wektorów wagowych  $w_1 = [1, 0, 0, 0]$  i  $w_2 = [0.25, 0.25, 0.25, 0.25]$  wynik mnożenia (wartość funkcji oceny)  $w_1^T x = w_2^T x = 1$ . Jednakże wprowadzenie kary  $l^2$  do funkcji kosztu powoduje, że preferowany będzie wektor  $w_2$ , gdyż  $\|w_2\| < \|w_1\|$ , a naszym celem jest minimalizacja funkcji kosztów. Widać zatem, że preferowane będą wektory wagowe o bardziej rozłożonych współczynnikach.

Składową danych ze wzoru 2.2 jest funkcją kosztu dla pojedynczego obrazka uśrednioną po wszystkich danych treningowych. Dla  $i$ -tego punktu treningowego możemy zdefiniować wiele różnych funkcji kosztu. Dla klasyfikatora typu wieloklasowa maszyna wektorów nośnych (*Multiclass SVM*) będzie to koszt "zawiasowy" (*hinge loss*, wzór 2.4), natomiast dla klasyfikatora typu *Softmax* koszt entropii krzyżowej (*cross-entropy loss*, przedstawiony na wzorze 2.5).

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta) \quad (2.4)$$

gdzie  $s_j = f(x_i, W)_j$  to  $j$ -ty element wektora ocen klas,  $s_{y_i}$  jest oceną uzyskaną przez prawdziwą klasę, a  $\Delta$  jest marigniem, który chcemy utrzymać pomiędzy wartością oceny klasy prawdziwej a wartościami ocen klas niepoprawnych.

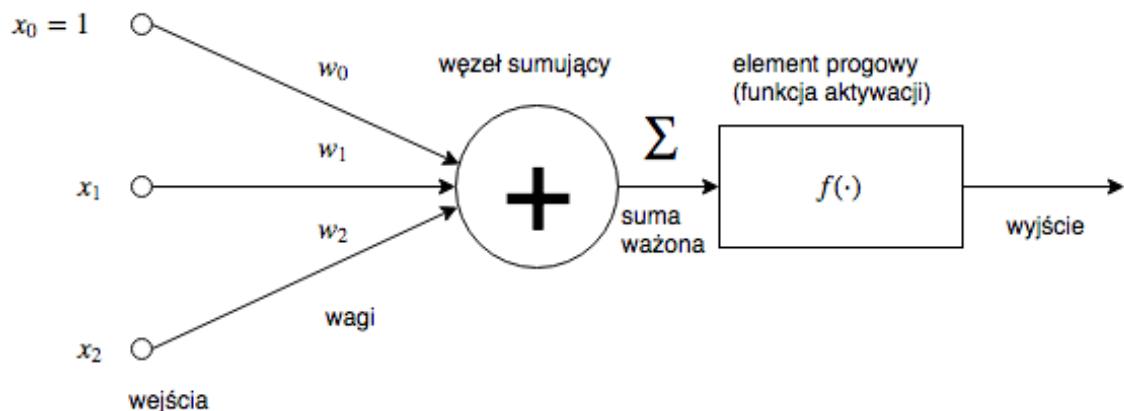
$$L_i = -\log \left( \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right) \quad \text{lub równoważnie} \quad L_i = -s_{y_i} + \log \sum_j e^{s_j} \quad (2.5)$$

gdzie  $s_j$  jak wyżej oznacza  $j$ -ty element wektora ocen klas  $s$ , tutaj interpretowany jako nieznormalizowany logarytm prawdopodobieństwa przynależności do danej klasy.

Możemy teraz znaleźć taką macierz wagową  $W$ , która minimalizuje wartość przyjętej funkcji kosztu, tym samym maksymalizując poprawność działania klasyfikatora. Jest to problem optymalizacji, który rozwiązuje najczęściej przy pomocy gradientowych metod optymalizacji, np. metody najszybszego spadku. [13]

Powodem, dla którego tyle miejsca poświęcono liniowemu klasyfikatorowi i pojęciom z nim związanymi jest fakt, że wszystkie te pojęcia mają również zastosowanie przy sieciach neuronowych.

### Sieci Neuronowe



Rys. 2.7. Pojedynczy 3-wejściowy neuron  
 Źródło: praca własna na podstawie [13, 20]

Zarówno zwierzęta, jak i ludzie lepiej radzą sobie z rozpoznawaniem obrazów niż współczesne komputery. Sieci neuronowe dlatego wzbudzają zainteresowanie, że możemy za ich pomocą częściowo naśladować ludzki mózg. Informacja w takich sieciach przetwarzana jest przez dużą ilość węzłów obliczeniowych oraz połączeń między nimi. Skoordynowane jednostki jednocześnie dokonują przetwarzania wszystkich lub większości sygnałów i danych wejściowych. Działanie sztucznych sieci neuronowych możemy porównać do rozproszonych systemów obliczeniowych. Elementarną jednostkę wykonującą obliczenia w takiej sieci nazywamy neuronem. Pojedynczy neuron dokonuje sumowania i nieliniowego przetwarzania sygnałów. Nierzadko możemy je przyrównać do elementów działających progowo, aktywujących się w momencie przekroczenia pewnego zadanego progu przez sumę sygnałów na wejściu.

Często neurony organizowane są w regularne struktury, zwykle możemy w owych strukturach wyróżnić poszczególne warstwy. Możliwe jest występowanie połączeń zwrotnych pomiędzy neuronami tej samej warstwy lub z różnych warstw. Każde połączenie pomiędzy neuronami charakteryzowane jest przez siłę jego oddziaływania zwaną również wagą. [20]

Schematycznie pojedynczy neuron możemy przedstawić jak na rys. 2.7

Pojedynczy neuron możemy również traktować jako klasyfikator liniowy, stąd obszerniejszy opis klasyfikatorów liniowych w 2.7. Matematyczny model neuronu od linowego klasyfikatora różni się jedynie zastosowaniem nieliniowości po operacji iloczynu skalarnego. Nieliniowość owa odpowiada za zdolność neuronu do prefeorowania pewnych liniowych obszarów jego wejścia (wartość funkcji aktywacji w okolicy 1) od innych (wartość funkcji aktywacji w okolicy 0). Tym samym stosując odpowiednią postać funkcji kosztu możemy sprowadzić zwykły neuron do klasyfikatora liniowego. [13]

Architektura sieci jest tym co odróżnia konkretne sieci od siebie. Na przykład sieci nie posiadające połączeń zwrotnych reagują na pobudzenie w sposób natychmiastowy i te właśnie sieci będą omawiane w dalszej części pracy. Natomiast sieci, które posiadają połączenia zwrotne, ustalają swoją odpowiedź dopiero po pełnym czasie, mają zatem swoją dynamikę określającą ich zachowanie w czasie. [20] Dalej rozważać będziemy jedynie sieci bez połączeń zwrotnych.

Sieci neuronne posiadają wiele zastosowań, wśród nich można wymienić:

- modelowanie zagadnień programowania liniowego
- rozpoznawanie pisma
- zastosowania w teorii sterowania
  - identyfikacja obiektów
  - neurosterowanie obiektem statycznymi
- sterowanie kinetyką robotów
- neuronowe sieci ekspertowe [20]

Pytanie o to, które z modyfikowalnych komponentów systemu uczącego się są odpowiedzialne za jego sukces lub porażkę i jakie zmiany w nich polepszą ich wydajność nazywane jest fundamentalnym problemem przypisania zasługi (*the fundamental credit assignment problem*). [17]. Innymi słowy do dziś nie wiadomo, co dokładnie jest czynnikiem sprawiającym, że jedna sieć będzie działać lepiej od drugiej. Badanie czy algorytm genetyczny jest w stanie poprawić jakość działania takiej sieci jest częściowo próbą znalezienia odpowiedzi na to pytanie.

### Konwolucyjne sieci neuronowe

Konwolucyjne sieci neuronowe (*Convolutional Neural Networks*) są podobne do opisanych w 2.8 sieci neuronowych. Tworzone są z neuronów, które posiadają swoje wagi, które zmieniają się w wyniku procesu uczenia. Każdy z neuronów otrzymuje sygnały na wejściu, dokonuje ich sumowania, a następnie może dokonać nieliniowego odwzorowania. Cała sieć wyraża pojedynczą, różniczkowaną funkcję odwzorowującą wartości jasności pikseli do zbioru klas. Sieci te projektowane są z założeniem, że będą przetwarzaly obrazy, stąd ich specyficzna architektura, która zostanie opisana poniżej.

Motywacją stojącą za poszukiwaniem innych architektur sieci neuronowych była bardzo szybko narastająca ilość wag do uczenia w przypadku przetwarzania obrazów za pomocą w pełni połączonych sieci neuronowych. Biorąc dla przykładu CIFAR-10 i obrazki o wymiarach  $32 \times 32 \times 3$  (wysokość, szerokość, 3 kanały kolorów), pojedynczy neuron znajdujący się zaraz za warstwą wejściową posiadałby  $3072 + 1$  wejście od stałej wartości (*bias*) wag do nauczenia. Dla tak małych obrazków liczba ta nie wydaje się być problematyczna, jednak podając na wejście obraz o wymiarach  $1024 \times 768 \times 3 = 2359296$  widzimy, że już pojedynczy neuron zajmowałby bardzo dużo pamięci. Zakładając, że każda waga przechowywana jest jako liczba zmiennoprzecinkowa o podwójnej precyzyji i przechowywana w 8 bajtach, uzyskujemy  $18874368 \text{ B} = 18\text{MB}$  na jeden neuron. Tak duża liczba parametrów jest nadmiarowa i może prowadzić do nadmiernego dopasowania.

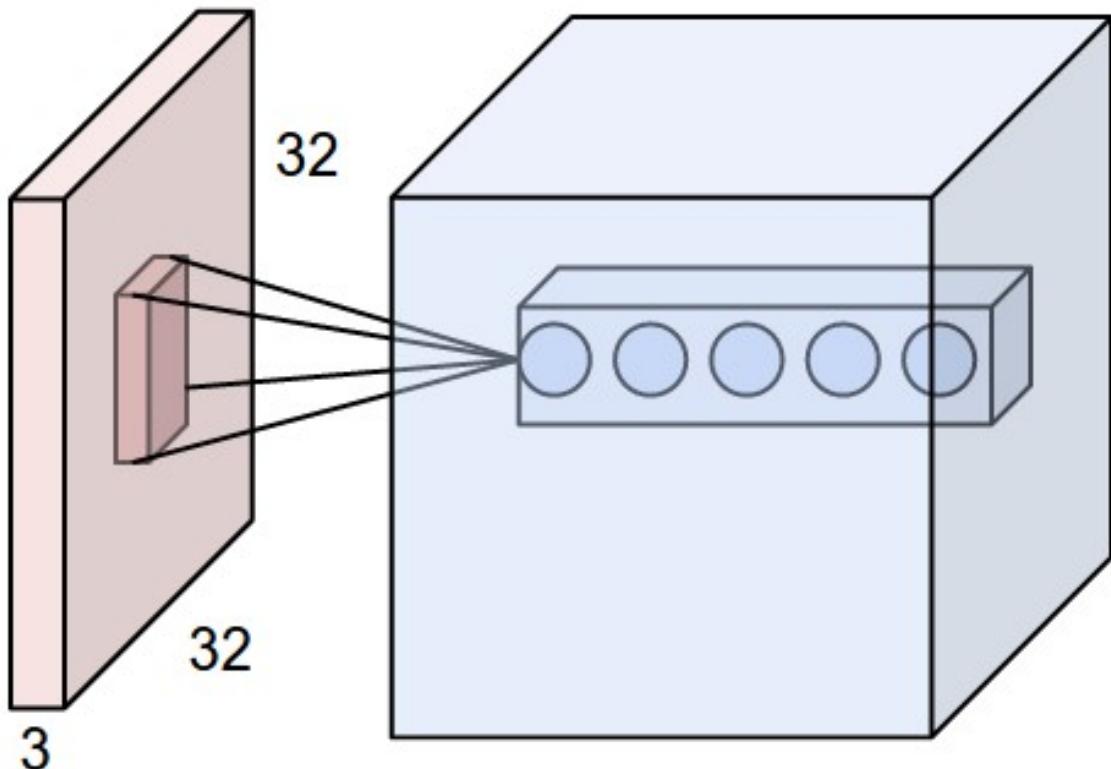
Konwolucyjne sieci neuronowe wykorzystują fakt, że danymi podawanymi na ich wejście są obrazy i nakładają ograniczenia na architekturę w bardziej rozsądny sposób. W szczególności, w przeciwieństwie do tradycyjnej sieci neuronowej, warstwy sieci konwolucyjnej posiadają neurony zorganizowane w 3 wymiarach: wysokość, szerokość i głębokość (rozumianej jako trzeci wymiar wejścia, nie

jako głębokość w sensie liczby warstw w sieci). W sieciach neuronowych każda warstwa przekształca trójwymiarowe wejście na trójwymiarowe wyjście przez różniczkowalną funkcję, która może lecz nie musi posiadać parametry.

W sieciach konwolucyjnych wyróżnia się kilka podstawowych rodzajów warstw, które następnie składa się w całą sieć. Trzy podstawowe typy to:

- warstwa konwolucyjna lub splotowa - oblicza wyjścia neuronów połączonych z lokalnymi fragmentami wejścia, wyznaczając iloczyn skalarny pomiędzy swoimi wagami a małym obszarem wejścia, do którego są połączone.
- warstwa agregująca - dokonuje operacji podpróbkowania wzdłuż wysokości i szerokości wejścia, zmniejszając wymiar danych na swoim wyjściu
- warstwa w pełni połączona - tak jak w klasycznych sieciach neuronowych, każdy neuron jest połączony z każdą jednostką wejścia

Przyjmować będziemy również, że warstwa aktywacji liniowej (*ReLU*, *Rectified Linear Unit*) jest "zaszyta" wewnętrz warstwy konwolucyjnej, na jej wylocie. Warstwa ta realizuje funkcję  $\max(0, x)$  w odniesieniu do każdego elementu na swoim wejściu. [13] Schematyczne organizację neurów w konwolucyjnej sieci neuronowej przedstawiono na rys. 2.8



Rys. 2.8. Schematyczne przedstawienie architektury konwolucyjnej sieci neuronowej  
Źródło: [13]

Na rysunku uwidoczniona została główna idea konwolucyjnych sieci neuronowych - najważniejszy parametr to liczba neuronów w warstwie, zwana również głębokością warstwy, która w tym przypadku wynosi 5. Każdy z tych neuronów posiada swój zestaw wag, który następnie jest splatany z obrazem wejściowym na całej jego szerokości i wysokości. Owe 5 filtrów "widzi" tylko swój mały kawałek obrazu, nazywany jego polem percepji (*receptive field*). Takie ograniczenie nazywane jest lokalną połączniwością (*local connectivity*). W wyniku operacji splotu otrzymywana jest mapa cech dla każdego neuronu, i w wyniku procesu uczenia neurony dobierają swoje wagę tak, by być wrażliwymi na cechy charakterystyczne dla danej klasy. Oprócz liczby filtrów z różnymi wagami, z których każdy splatany jest przez na zasadzie ruchomego okna, dodatkowymi parametrami cechującymi taką sieć są:

- przeskok (*stride*) - okno można przesuwać konsekwentnie piksel po pikselu po całym obrazie, można jednak przesuwać o większą ilość pikseli, owocuje to jednak zmniejszeniem wymiaru obrazu wyjściowego
- dopełnianie zerami (*zero padding*) - w przypadku skrajnych pikseli np. o wymiarach 3x3 przesuwane jest po nieistniejących w obrazie pikselach, które najczęściej zastępowane są zerami. Rozmiar ramki złożonej z zer jest parametrem, który pozwala nam kontrolować rozmiar wyjścia.

Rozmiar obrazu po przejściu przez warstwę tak zdefiniowaną można obliczyć ze wzoru 2.6

$$(W - F + 2P)/S + 1 \quad (2.6)$$

gdzie W to wymiary obrazu wejściowego, F to rozmiar pola percepcji neruonu, P to liczba dopełnianych zer a S to przeskok z jakim przesuwamy okno.

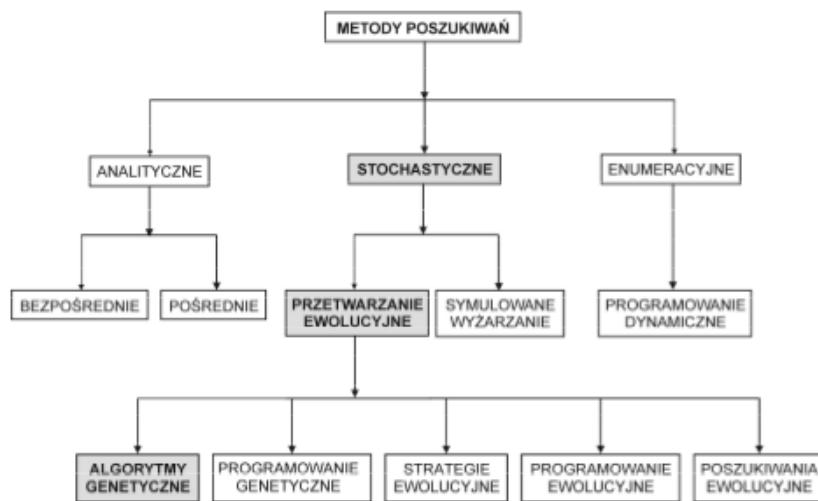
Co do warstwy agregującej, również wykonuje ona operacje na małym polu wejściowym, dając w wyniku jeden skalar - np. z pola 2x2 wybierana jest maksymalna wartość lub średnia ważona wszystkich wartości. Operacja taka ma na celu zwiększenie kontroli nad nadmiernym dopasowaniem przez algorytm, i w wyniku daje pomniejszony wymiar obrazu (mapy) wyjściowej, np. przeprowadzenie agregacji 2x2 na warstwie 32x32 zaowocuje obrazem o wymiarach 16x16 na wyjściu.

Warstwa w pełni połączona zazwyczaj używana jest na końcu i zwykle znajduje się w niej tyle neuronów, ile klas, a ich wyjście mówi nam (w przypadku zastosowania odpowiedniej funkcji kosztu przy uczeniu sieci) o prawdopodobieństwie przynależenia danego obrazka do danej klasy.

Sposród wymienionych wyżej parametrów, kontrolowanym przez algorytm genetyczny parametrem będzie głębokość 4 warstw.

### **Algorytm genetyczny**

Algorytm genetyczny jest kolejnym opisywanym narzędziem sztucznej inteligencji, które powstało dzięki obserwacji natury i jej sposobów działania. Wywodzi się on z genetyki, czyli z badania dziedziczności i zmienności organizmów. Pierwsze badania z tej dziedziny w XIX wieku przeprowadzał Gregor Johann Mendel. W ich wyniku powstało pojęcie genu - abstrakcyjnej jednostki dziedziczności oraz alleli - różnych odmian tego samego genu. Dalej możemy mówić o chromosomach, czyli strukturach znajdujących się wewnętrz jąder komórek zwierzęcych i roślinnych, które zawierają geny. Chromosomy zbudowane są z kwasu dezoksyrybonukleinowego (DNA), które koduje całą informację genetyczną danego osobnika. Na podstawie obserwacji sposobu dziedziczenia i rozmnażania się komórek, procesu ewolucji zaproponowano algorytm genetyczny, który w istocie jest stochastyczną metodą optymalizacji, jak przedstawiono na rys. 2.9.



Rys. 2.9. Podział metod optymalizacji  
Źródło: [5]

Podstawowe cechy algorytmu genetycznego:

1. poszukiwania prowadzone są w wielu punktach
2. zadanie optymalizacji sprametryzowane w sposób zakodowany
3. nowe rozwiązania tworzone i wybierane sa przy pomocy metod probabilistyczno-ewolucyjnych
4. brak ograniczeń na przestrzeń poszukiwań - funkcja celu może być dowolna, nieznana
5. prostota implementacji i uniwersalność

Główne ze względu na punkt 4 wybrano tę metodę do optymalizacji struktury konwolucyjnych sieci neuronowych, gdyż funkcja skuteczności sieci w zależności od jej struktury jest nieznana.

Istotnymi pojęciami w AG są pojęcia genotypu i fenotypu. Genotyp rozumiemy jako zakodowane parametry reprezentujące danego osobnika. Poniżej wymieniono kilka podstawowych sposobów kodowania parametrów w AG:

- bialleliczne
- Gray'a
- logarytmiczne
- trialleliczne
- multialleliczne
- wielopoziomowe
- całkowitoliczowe

Fenotyp jest uzupełnieniem tych zakodowanych cech.

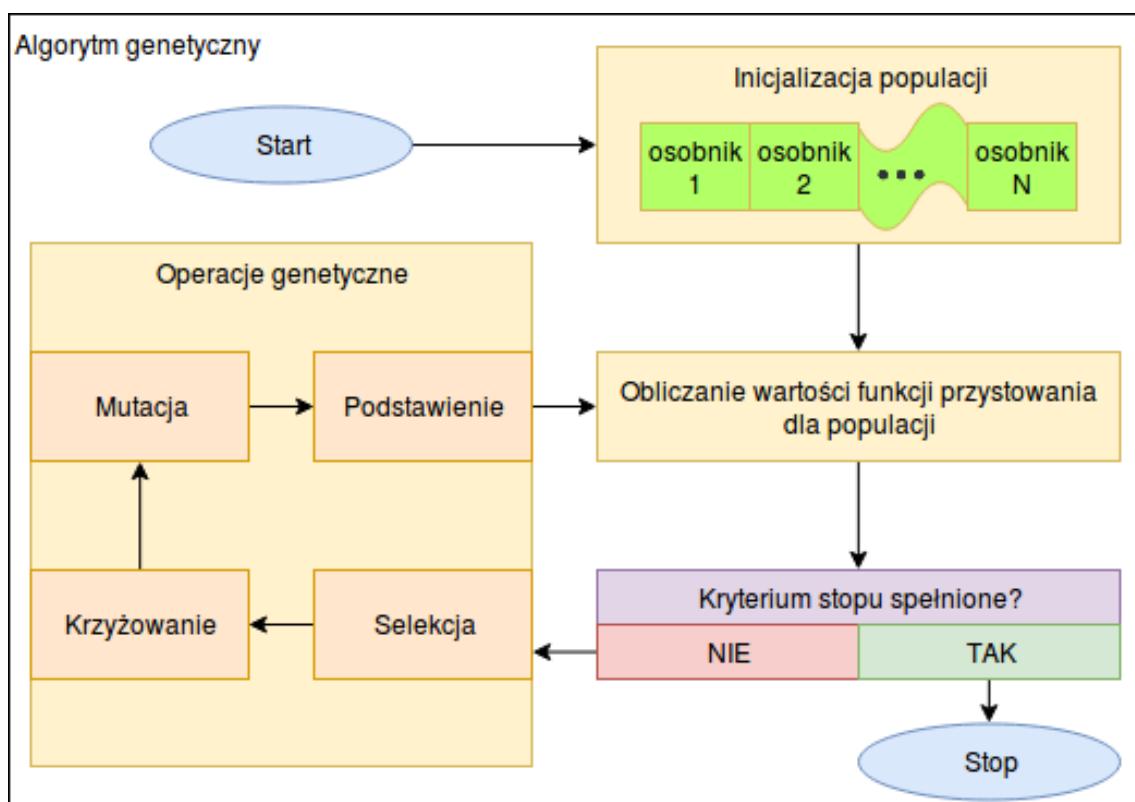
Podstawowe operacje algorytmu genetycznego zwane operacjami genetycznymi oraz kilka ich realizacji wymieniono poniżej:

- selekcja:
  - metodą proporcjonalną
  - metodą ze stochastycznym doborem resztowym
  - metodą turniejową
  - metodą rangową
  - metodą progową
- krzyżowanie (pominięto sposoby krzyżowania dla kodowania całkowitolicbowego)
  - jednopunktowe
  - wielopunktowe
  - jednorodne (równomierne)
  - arytmetyczne
  - mieszane
- mutacja:
  - genotypowa
  - fenotypowa
    - \* równomierna
    - \* nierównomierna
  - inwersja
  - wstawianie
  - przenoszenie
  - wzajemna wymiana

- podstawienie:
  - reprodukcja pełna
  - reprodukcja częściowa
    - \* usuwanie najgorszych osobników - elityzm
    - \* usuwanie najbardziej podobnych do potomstwa osobników - metoda ze ściskiem
    - \* usuwanie losowo wybranych osobników

[5]

Algorytm genetyczny początkowo wytwarza losową populację osobników (rozwiązań). Następnie dla każdego z osobników wyznaczana jest wartość funkcji przystosowania, która ma być zmaksymalizowana. Następnie, spośród ocenionych już osobników wybierana jest pula rodzicielska przy pomocy jednej z metod selekcji. Wewnątrz wybranej puli rodzicielskiej z pewnym prawdopodobieństwem nazywanym prawdopodobieństwem krzyżowania dochodzi do krzyżowania, czyli powstania dwóch nowych potomków z dwóch rodziców. Następnie, wszystkie osobniki z pewnym prawdopodobieństwem zwanym prawdopodobieństwem mutacji są modyfikowane. Ostatecznie, całość lub część starej populacji jest wymieniona według jednej ze strategii reprodukcji. Powyższe działania powtarzane są tak długo, aż nie otrzymamy satysfakcyjnego rozwiązania, np. rozwiązanie nie poprawi się przez 100 kolejnych iteracji. Działanie algorytmu genetycznego można schematycznie przedstawić jak na rysunku 2.10



Rys. 2.10. Schemat działania algorytmu genetycznego  
Źródło: praca własna

## Opis rozwiązania

### Redukcja problemu

Jak już wspomniano w rozdziale 2, uczenie głębszych sieci neuronowych w celu klasyfikacji obrazów dla dużych danych wejściowych jest zadaniem czasochronnym. Jednym ze sposobów redukcji tego czasu jest stosowanie architektury konwolucyjnej (opisanej w 2.9), która znacznie redukuje liczbę wag do nauczenia, a przy tym i czas obliczeń. Nauczenie pojedynczej sieci konwolucyjnej wymaga wielu epok uczenia, najczęściej wykorzystując równoległe obliczenia na kartach graficznych, np. wykorzystując bibliotekę CUDA w połączeniu z biblioteką TensorFlow. Osiągnięcie skuteczności klasyfikacji na poziomie 86% dla zbioru CIFAR-10 dla przykładowej sieci zajmuje ok.  $t_{cnn} = 3$  godzin dla typowej karty graficznej Nvidia GeForce GTX850M. [1]

W przypadku algorytmu genetycznego, jedna konwolucyjna sieć neuronowa jest jednym osobnikiem charakteryzowanym przez jego fenotyp, opisany w 3.2. Aby algorytm genetyczny skutecznie przeszukiwał przestrzeń poszukiwań, potrzebna jest stosunkowo duży rozmiar populacji  $N$ . Jeśli wyznaczenie funkcji przystosowania byłoby jednoznaczne z wyznaczeniem skuteczności sieci po pełnym uczeniu, całkowity czas dla jednej iteracji dla  $N = 100$ :

- Przy sekwencyjnym uczeniu każdego osobnika wyniosłyby  $T_s = Nt_{cnn} = 300$  h = 12.5 dnia.
- Przy równoległym uczeniu każdego osobnika na  $m = 10$  węzłach obliczeniowych wyniesie  $T_p = \frac{T_s}{m} = 30$  h = 1.25 dnia.

Narzucając (oprócz kryterium stopu) maksymalną liczbę iteracji równą  $I = 100$ , w pesymistycznym przypadku dla obliczeń równoległych przy  $m = 10$  całkowity czas obliczeń wyniosłyby  $T_c = IT_p = 125$  dni. W związku z ograniczonym czasem na przeprowadzenie badań oraz minimalizację kosztów obliczeń, czas ten musiał zostać znacznie zredukowany.

Jako rozwiązanie tego problemu zaproponowano:

1. uczenie małych sieci, o ustalonych ramach i zmienności struktury wyznaczanej przez zmienną liczbę filtrów w warstwach konwolucyjnych (takich jak opisane w 3.2).
2. przyjęcie jako celu optymalizacji skuteczność sieci już po jednej epoce uczenia sieci.

Dzięki przyjęciu takiego podejścia czas wyznaczania wartości funkcji przystosowania pojedynczego osobnika redukuje się do średniej wartości (wyznaczonej eksperymentalnie) około  $t_{cnn} \approx 2.5$  minut, co dla  $m = 20$  sprawdza czas maksymalny działania algorytmu do  $T_c \approx 21h$ . Na tej zasadzie skonstruowano eksperyment opisany w 4.4.

Po uczeniu sieci przez tak krótki czas, można sprawdzić czy wynik osiągany przez sieć już po jednej epoce uczenia ma związek z wynikiem uzyskanym po uczeniu dłuższym, np. dziesięciu epokach. Proponowanym sposobem sprawdzenia powyżej sformułowanej hipotezy jest wzięcie najlepszego, środkowego i najgorszego osobnika z każdej iteracji algorytmu genetycznego i uczenie ich dłużej, np. przez 10 epok. Ilość zmian kolejności najlepszy-środkowy-najgorszy dla przypadków 1- i 10- epokowego uczenia w stosunku do całkowitej liczby iteracji definiuje stopień niepewności hipotezy, że istnieje wspominany związek.

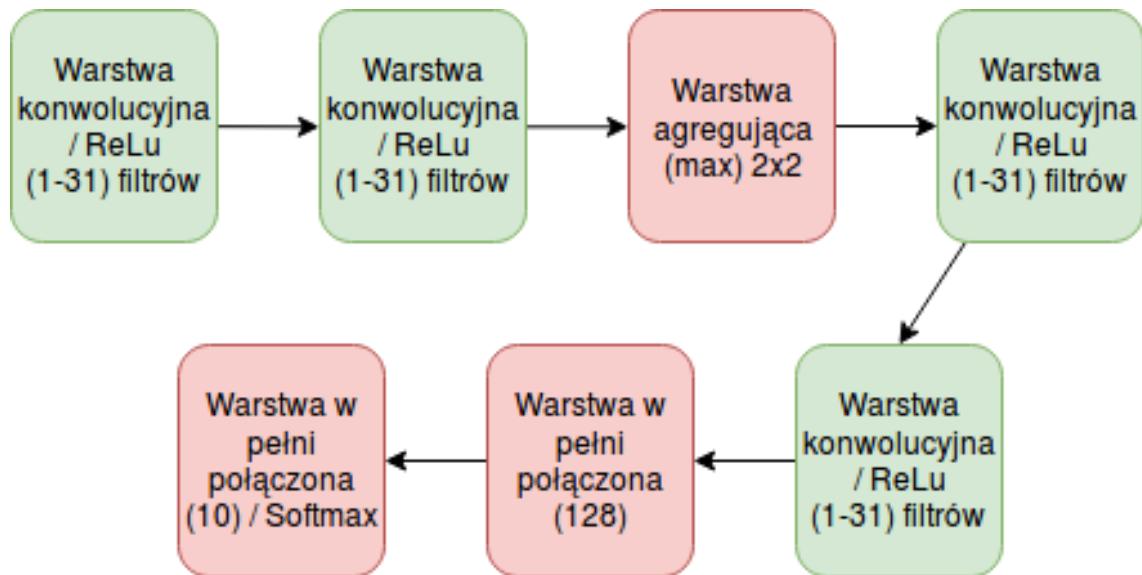
Wybór 3 osobników ze 100 w każdej iteracji zmniejsza liczba wyznaczeń wartości nowej funkcji przystosowania z  $N * I = 10000$  do  $3 * I = 300$ , czyli ponad 33-krotnie, co pozwoliłoby uczyć przez 33 epoki w tym samym czasie. W celu dalszej redukcji czasu i kosztów obliczeń zdecydowano się na nowe obliczenia dla 10-epok.

Należy pamiętać, że w tym eksperymencie nie szukamy nowych rozwiązań, a sprawdzamy inną funkcję przystosowania dla wybranego ich podzbioru.

### Sieć neuronowa jako osobnik algorytmu genetycznego

Przykładową sieć neuronową powstającą w wyniku działania algorytmu przedstawiono na rys. 3.1. Na rysunku widać dwie następujące po sobie warstwy konwolucyjne, każda ze zmienną liczbą filtrów, dobieraną przez algorytm genetyczny. Następująca po niej warstwa agregująca zmniejsza wymiar danych wejściowych dla kolejnych dwóch warstw konwolucyjnych, dla których liczba filtrów również jest dobierana przez algorytm. Po niej następuje warstwa w pełni połączonych 128 neuronów, za którą znajduje się warstwa 10 neuronów, każdy odpowiadający jednej klasie obrazka ze zbioru CIFAR-10.

Dzięki takiej architekturze osobnik może zostać sprametryzowany jako lista czterech liczb całkowitych, nazywana dalej fenotypem, które można zakodować na 5 bitach każdą, przechodząc w ten sposób do genotypu. Binarne kodowanie konieczne jest do wybranego sposobu krzyżowania osobników, opisanego w 3.3. Zaproponowana sieć jest arbitralnie dobraną siecią, dla której bloki przedstawione na czerwono są ustalone, a ilość filtrów w blokach przedstawionych na zielono jest zmienna.



Rys. 3.1. Przedstawienie pojedynczego osobnika  
Źródło: praca własna

Odgórne osiągi zbioru sieci ograniczone są przez elementy ustalone, tj. czerwone elementy na rys. 3.1, stałą liczbę warstw konwolucyjnych (4) oraz granice w jakich zmieniają się liczby filtrów w tychże (1-31). Zadaniem algorytmu genetycznego będzie maksymalizacja jakości klasyfikacji obrazów tak opisanych sieci po jednej epoce uczenia. Przy tak dobranych parametrach osobnika możliwe jest wystąpienie zjawiska nadmiernego dopasowania do zbioru uczącego przez otrzymane sieci, w związku z brakiem zastosowania środków zapobiegawczych (takich jak dropout).

Funkcja przystosowania dla pojedynczego osobnika, jak już wspomniano w 3.1, została określona jako skuteczność sieci po jednej epoce uczenia. Nic nie stoiaby jednak na przeszkodzie, aby dodać inne kryteria do funkcji przystosowania, np. gdyby zależało nam na uzyskaniu sieci która zajmuje mało pamięci moglibyśmy odjąć od funkcji przystosowania odjąć odpowiednią karę proporcjonalną do sumy ilości wszystkich filtrów w warstwach.

### **Operacje genetyczne algorytmu genetycznego**

Spośród wymienionych w 2.10 operacji genetycznych należało wybrać, jakie zaimplementować w programie realizującym algorytm genetyczny.

#### **Selekcja**

Wybranym w tej pracy sposobem selekcji jest sposób proporcjonalny, zwany również ruletkowym. Każdy z osobników przypisany zostaje do obszaru ruletki, którego rozmiar jest proporcjonalny do wartości funkcji przystosowania tego osobnika. Następnie  $N$ -krotnie kręci się ruletką, a pole, na które wypadnie, determinuje który z osobników przechodzi do puli potomków. W ten sposób lepiej przystosowane osobniki mają większą szansę przejścia dalej i przekazania swoich genów. W przypadku długotrwałego obliczania funkcji przystosowania dodatkową zaletą takiego podejścia jest większa szansa na pojawienie się takich samych osobników (które nie zostaną poddane krzyżowaniu ani mutacji) w kolejnej iteracji algorytmu. Z punktu widzenia efektywniejszego przeszukiwania przestrzeni dostępnych rozwiązań nie jest to zaletą, jednakże pozwala oszczęścić obliczeń dla już obliczonych punktów.

Tutaj metoda proporcjonalna została zaimplementowana w następujący sposób:

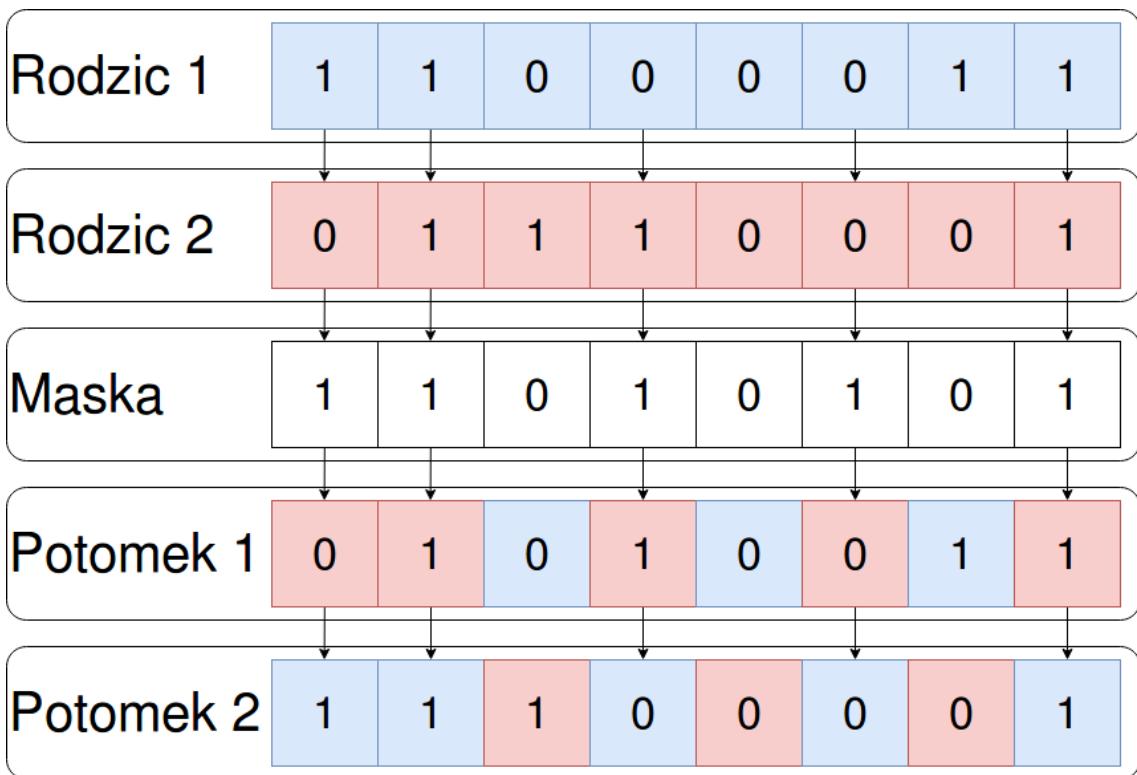
Powtórz  $N$ -krotnie:

1. Sumowane są wartości funkcji przystosowania  $S(x) = \sum_{i=1}^N f(x_i)$  dla każdego  $x_i$  w populacji

2. Losowana jest losowa liczba z przedziału  $r = (0, S(\mathbf{x}))$
3. Znajdowane jest najmniejsze  $i$ , dla którego w danej sekwencji osobników  $\sum_{i=1}^N x_i > r$

#### Krzyżowanie

Przy implementacji algorytmu genetycznego wybrano krzyżowanie jednorodne. Dla pary chromosomów rodzicielskich, które są w istocie dwoma ciągami binarnymi o jednakowej długości, losowana jest binarna maska o tej samej długości. Następnie następuje zamiana miejscami bitów oznaczonymi jako 1 w wylosowanej masce, w wyniku czego powstają dwa nowe chromosomy - potomkowie. Działanie to w sposób schematyczny przedstawiono na rys. 3.2. Rozważając poniższy przykład jako dwie warstwy konwolucyjne zakodowane na 4-bitach każdy, z rodziców [12, 3] oraz [7, 1] otrzymujemy z daną maską dwie nowe sieci z warstwami: [5, 3] oraz [14, 1].



Rys. 3.2. Krzyżowanie jednorodne  
 Źródło: praca własna

#### Mutacja

Sposób mutacji wykorzystany w tej pracy tożsamy jest ze zmianą wartościowo-równomierną. Dokonywana jest ona na fenotypie osobnika. Jeżeli mutacja danego osobnika zachodzi, to:

- Losowo wybierana jest liczba ze zbioru  $\{0, 1, 2, 3\}$ , która determinuje, który parametr będzie zmieniany
- Następuje wylosowanie nowej wartości dla wybranego parametru z dopuszczalnego zbioru liczb całkowitych  $\{1..31\}$ , który zastępuje starą wartość

#### "Naprawianie" osobników

Istnieje pewne prawdopodobieństwo, że w wyniku krzyżowania otrzymamy osobnika, dla którego jeden lub więcej parametrów przyjmie wartość zero. Rozwiązać ten problem można na dwa sposoby:

1. Na poziomie algorytmu genetycznego - zamieniając otrzymane 0 losową lub najbliższą (1) dopuszczalną wartością

2. Na poziomie konstruowania sieci neuronowej - jeżeli warstwa zawiera 0 filtrów to jest pomijana w sekwencyjnym budowaniu modelu sieci.

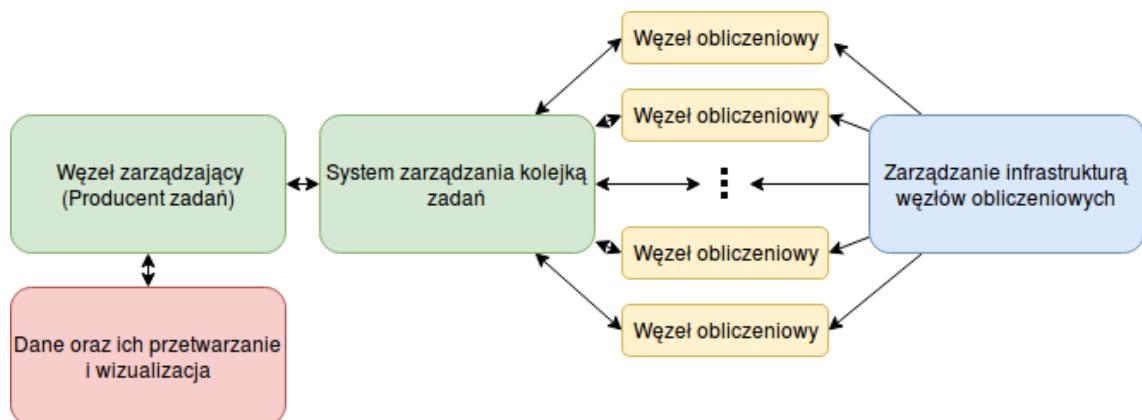
W pracy zdecydowano się zastosować się podejście 1 z zastępowaniem 0 przez 1, ze względu na prostotę implementacyjną.

### *Substytucja*

Zastosowana w algorytmie technika substytucji osobników rodzicielskich przez potomków jest strategią z reprodukcją częściową. W celu zmniejszenia ilości obliczeń potomkowie nie są oceniani do momentu, kiedy staną się rodzicami w następnym pokoleniu. Z tego powodu zastosowano połączenie strategii elitarnej (pozostawienie najlepszych osobników z poprzedniej iteracji) z zastępowaniem losowo wybranych osobników (losowo wybierane są nowe osobniki z puli potomków). Pozwala to dodatkowo zaoszczędzić obliczeń dla przeniesionej z poprzedniej iteracji elity przy jednoczesnym zachowaniu różnorodności genetycznej pochodzącej od losowo wybranych potomków. Dodatkowo, w stosunku do pełnej reprodukcji (z wymianą całej populacji), algorytm szybciej znajduje lepsze rozwiązania.

### *Architektura eksperymentu*

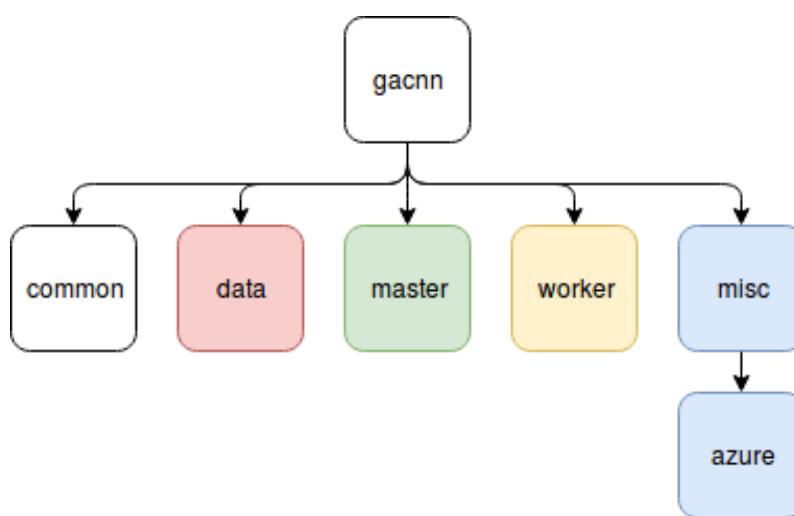
W toku przygotowywania eksperymentów powstał system napisany w języku Python[19]. Najważniejsze jego elementy zostały wyróżnione na rys. 3.3



Rys. 3.3. Całokształt systemu przeprowadzania eksperymentu

Źródło: praca własna

Każdy z powyżej przedstawionych elementów zostanie omówiony dalej. Poniżej, na rys. 3.4 przedstawiono strukturę katalogów projektu:



Rys. 3.4. Struktura katalogów projektu systemu eksperymentu

Źródło: praca własna

Całość projektu zorganizowano w katalogi tak, aby zgrupować pliki źródłowe powiązane z konkretnymi blokami funkcjonalni z rys. 3.3. Bloki oznaczone na obydwu rysunkach tym samym kolorem są swoimi odpowiednikami. W związku z tym:

- katalog *gacnn* jest korzeniem, również nazwą repozytorium kodu źródłowego. Jego nazwa jest abrewiacją angielskich odpowiedników nazw algorytm genetyczny i konwolucyjna sień neuronowa.
- katalog *common* zawiera elementy potrzebne możliwie przez każdy z innych modułów
- katalog *data* przechowuje zebrane przez węzeł zarządzający dane oraz skrypty służące do ich przetwarzania i wizualizacji
- katalog *master* zawiera wszystkie programy, które mogą być węzłem zarządzającym, oraz system zarządzania kolejką zadań
- katalog *worker* zawiera oprogramowanie uruchamiane na każdym z węzłów obliczeniowych
- katalog *misc* zawiera pomocnicze skrypty służące do tworzenia i przygotowywania węzłów obliczeniowych
- katalog *azure* zawiera skrypty j.w. przeznaczone konkretnie do tworzenia instancji węzłów obliczeniowych na platformie Microsoft Azure

#### *Elementy wspólne*

Wspólne elementy autorskiego kodu używane zarówno przez węzeł zarządzający, jak i przez węzły obliczeniowe, to definicje dwóch klas:

- Klasa *Job* - opisuje zadane przekazywane przez węzeł zarządzający do systemu kolejkowego i dalej do węzła obliczeniowego. W istocie swej jest to struktura przechowująca 3 elementy:
  - osobnika (reprezentowanego przez obiekt klasy *Individual*), dla którego należy wyznaczyć funkcję przystosowania
  - liczbę epok uczenia sieci neuronowej
  - ziarno dla generatora pseudolosowego
- Klasa *Individual* - opisuje pojedynczego osobnika, tj. pojedynczą strukturę sieci neuronowej. Przechowuje ona 4 informacje:
  - Fenotyp osobnika - lista 4 liczb opisujących liczbę filtrów w odpowiednich warstwach sieci
  - Przystosowanie osobnika - wartość funkcji przystosowania osobnika. Dla algorytmu optymalizującego strukturę konwolucyjnej sieci neuronowej jest to skuteczność klasyfikacji zbioru testowego przez nauczoną sieć.
  - Ostateczna uzyskana wartość funkcji kosztu (*loss function*) po uczeniu - w celu informacyjnym
  - Historię uczenia sieci - przechowuje wartości przystosowania i funkcji kosztu po każdej epoce uczenia sieci, przechowywane w celach informacyjno-diagnostycznych

#### *Węzeł zarządzający*

Węzłem zarządzającym nazywany będzie program, który korzysta z opisanego w 3.4.3 systemu kolejkowego do rodzielania swoich zadań i odbierania wyników. Jego głównym zadaniem jest typowanie osobników, dla których ma zostać wyliczona funkcja przystosowania. Dla potrzeb wspomnianych w 3.1 funkcji przystosowania dla uczenia 1- i 10- epokowego konieczne było stworzenie dwóch programów zarządzających:

1. Węzeł typujący osobniki do wyliczenia na podstawie działania algorytmu genetycznego dla uczenia 1-epokowego - plik *gentic\_algorithm.py*
2. Węzeł typujący najlepsze, środkowe i najgorsze osobniki z każdej iteracji algorytmu genetycznego dla uczenia 10-epokowego - plik *full\_learning.py*

Obydwa z wyżej wymienionych węzłów korzystają z pomocniczych funkcji do zapisywania odebranych od węzłów obliczeniowych danych - plik `csv_handling.py`.

Węzeł opisany powyżej w punkcie 1 został napisany tak, by możliwie łatwo można było regulować parametry algorytmu. Poniżej przedstawiono ich listę:

- liczba osobników  $N$
- wymiarowość przestrzeni poszukiwań - liczba warstw konwolucyjnych  $n$
- zakres przestrzeni poszukiwań - liczba bitów przypadających na kodowanie jednej warstwy  $m$
- prawdopodobieństwo krzyżowania -  $p_c$
- prawdopodobieństwo mutacji -  $p_m$
- maksymalna liczba iteracji
- maksymalna liczba iteracji bez poprawy
- liczba najlepszych osobników przechodzących do kolejnego pokolenia -  $N_k = 30$
- minimalna poprawa  $\epsilon$
- ziarno generatora pseudolosowego - domyślnie 1337

Po ustawieniu wyżej wymienionych parametrów, program działa następująco:

1. Ustawia zadane ziarno generatora losowego. Przyczny, dla których to robi, opisano w 3.5.
2. Jeśli nie istnieje, tworzy plik `.csv` do przechowywania danych, gdzie każdy wpis zawiera:
  - Numer iteracji
  - Genotyp osobnika w formie zakodowanej
  - Wartość funkcji przystosowania
  - Wartość funkcji kosztu
3. Jeżeli istnieje plik `.csv` z danymi z niedokończonego działania programu, wczytuje je.
4. Jeżeli nie wczytano żadnych danych, następuje inicjalizacja populacji. Losowo generowanych jest  $N$  osobników.
5. Dopóki nie osiągnięto kryterium stopu (maksymalna liczba iteracji bez poprawy wyniku lub maksymalna liczba iteracji osiągnięte)
  5. 1. Przygotowuje listę zadań dla systemu kolejkowego - dla każdego osobnika tworzy jedno zadanie, które w jednym zdaniu brzmi: "Ucz tego osobnika przez 1 epokę z takim ziarnem generatora pseudolosowego".
  5. 2. Oblicza wartość funkcji przystosowania dla całej populacji korzystając z metody `evaluate(jobs)` systemu kolejkowania zadań.
  5. 3. Sortuje osobniki według wartości funkcji przystosowania
  5. 4. Zapisuje dane z tej iteracji do pliku `csv`
  5. 5. Sprawdza najlepszy wynik i czy w tej iteracji nastąpiła poprawa - jeśli nie, zwiększa licznik iteracji bez poprawy
  5. 6. Za starą populację podstawia nową uzyskaną w wyniku:
    5. 6. 1. selekcji
    5. 6. 2. krzyżowania
    5. 6. 3. mutacji
    5. 6. 4. "naprawiania" osobników
    5. 6. 5. substytucji

Wszystkie z powyższych odbywają się jak opisano w 3.3

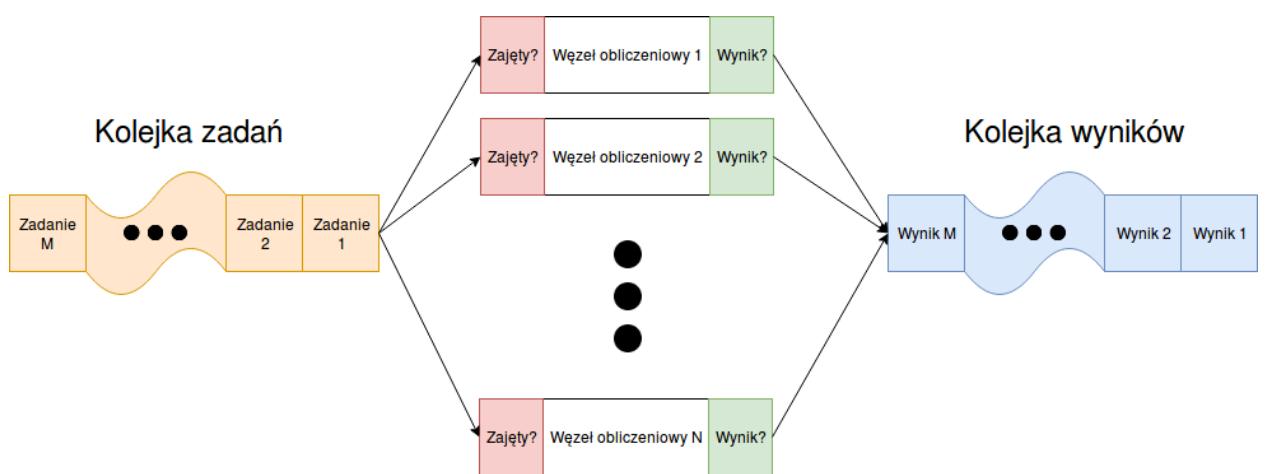
W wyniku działania powyższego skryptu otrzymujemy plik csv zawierający wszystkie osobniki z wszystkich iteracji działania algorytmu genetycznego. Dane te są przetwarzane dalej w celu wizualizacji, jak również przez skrypt *full\_learning.py*, który również jest węzłem zarządzającym, a działa następująco:

1. Wczytuje osobniki zapisane w pliku .csv przez algorytm genetycznych
2. Z każdej iteracji wybiera najlepszego, średkowego i najgorszego osobnika
3. Dla każdego wybranego osobnika tworzy zadanie, które w jednym zdaniu brzmi: "Ucz tego osobnika przez 1 epokę z takim ziarnem generatora pseudolosowego".
4. Oddelegowuje zadania do systemu kolejkowania zadań.
5. Zbiera wyniki i zapisuje je do innego pliku.

Ten skrypt wyznacza inną funkcję przystosowania dla tych samych osobników - skuteczność po 10 epokach uczenia, w porównaniu do 1-epokowego uczenia.

#### *System kolejkowania zadań*

W celu maksymalnego przyspieszenia obliczeń, dokonano zrównoleglenia obliczania funkcji przystosowania dla każdego osobnika. Ze względu na ograniczoną liczbę jednostek obliczeniowych mogących jednocześnie przetwarzać żądania, zaprojektowano system kolejkowy. Schemat tego systemu przedstawiono na rys. 3.5



Rys. 3.5. System kolejkowania obliczeń  
Źródło: praca własna

Implementacja systemu kolejkowego zawarta jest w pliku *queue\_handling.py*. Zawiera on następujące elementy:

- Interfejs *Worker* będącą reprezentacją pojedynczego węzła obliczeniowego.
- Klasę *RemoteWorker* dziedziczącą po klasie *Worker*, służącą do delegowania zadań do zdalnej maszyny wirtualnej wykonującej obliczenia.
- Klasę *LevyWorker* dziedziczącą po klasie *Worker*, lokalnie wyznaczającą wartość funkcji Levego, używaną przy testowaniu działania algorytmu genetycznego jak opisano w 4.2.
- Klasę *WorkManager* będącą właściwą implementacją systemu kolejkowania.

Interfejs *Worker* zawiera następujące metody:

- *is\_available()* - mówi, czy dany węzeł może przyjmować teraz zadania
- *assign\_job(job)* - przypisuje do danego węzła przekazane jako argument zadanie do obliczenia
- *has\_result()* - odpytuje węzeł, czy posiada już wynik obliczeń

- *get\_result* - zwraca odebrany od węzła wynik obliczeń
- *free\_worker()* - zwalnia węzeł oznaczając go jako gotowy do rozpoczęcia dalszych obliczeń
- *get\_current\_job()* - zwraca aktualnie zlecone węzowi zadanie

Właściwą implementacją systemu kolejkowego jest klasa *WorkManager*. System kolejkowy przechowuje wewnętrznie listę dostępnych węzłów obliczeniowych. Jedyną publiczną metodą tej klasy jest funkcja *evaluate(jobs)*, która jako argument przyjmuje listę zadań do wykonania. Dla przypadku obsługi węzłów zdalnych system kolejkowy:

1. Wyszukuje duplikaty wśród otrzymanych zadań i zauważa listę zadań do unikalnego zbioru
2. Dopóki liczba uzyskanych wyników jest mniejsza od liczby zleconych zadań:
  2. 1. Sprawdza plik *workers* z listą adresów IP potencjalnych węzłów
  2. 2. Próbuje połączyć się z węzłami zgodnie z procedurą opisaną w 3.4.4.
  2. 3. Dodaje każdy zgłoszający się węzeł do listy węzłów i oznacza go jako wolny.
  2. 4. Rodziela zadania każdemu z węzłów, dla każdego węzła z listy:
    2. 4. 1. Sprawdza czy ten węzeł jest oznaczony jako wolny i czy są dostępne zadania. Jeśli nie, przechodzi do kolejnego węzła i powtarza obecny krok.
    2. 4. 2. Pobiera zadanie z puli dostępnych zadań
    2. 4. 3. Próbuje przypisać pobrane zadanie do tego węzła (może się to nie udać np. z powodu utraty połączenia sieciowego).
    2. 4. 4. Jeśli powyższy krok się nie powiodł, przywraca zadanie do puli zadań, a niedziałający węzeł jest usuwany z listy węzłów.
  2. 5. Odpytuje każdy zajęty węzeł o wyniki w następujący sposób:
    2. 5. 1. Odpytuje węzeł czy wynik jest dostępny. Jeśli nie, przechodzi do kolejnego węzła i powtarza obecny krok.
    2. 5. 2. Pobiera od węzła jego wynik i zadanie. Zapisuje je do słownika, gdzie zadanie jest kluczem, a wynik wartością.
    2. 5. 3. Oznacza węzeł jako wolny.
    2. 5. 4. Jeśli któryś z powyższych kroków się nie powiodł, przywraca zadanie węzła do puli i usuwa go z listy węzłów.
  2. 6. W celu zmniejszenia obciążenia sieci przy komunikacji z węzłami, oczekuje sekundę przed kolejną iteracją.
3. Przywraca usunięte w punkcie 1 duplikaty, przepisując odpowiednie wyniki z policzonych zadań ze słownika zadanie-wynik.

W przypadku testu dla lokalnie obliczajone wartości funkcji *Levego* opisanego w 4.2 system ten, z pewnymi modyfikacjami, również może zostać wykorzystany. Obliczenia wykonywane są lokalnie i dostępne są natychmiast po zleceniu zadania, w związku z czym:

- lista węzłów zawiera tylko jeden węzeł *LevyWorker* i nie jest w żaden sposób modyfikowana
- pominięte zostaje opóźnienie odciążające sieć

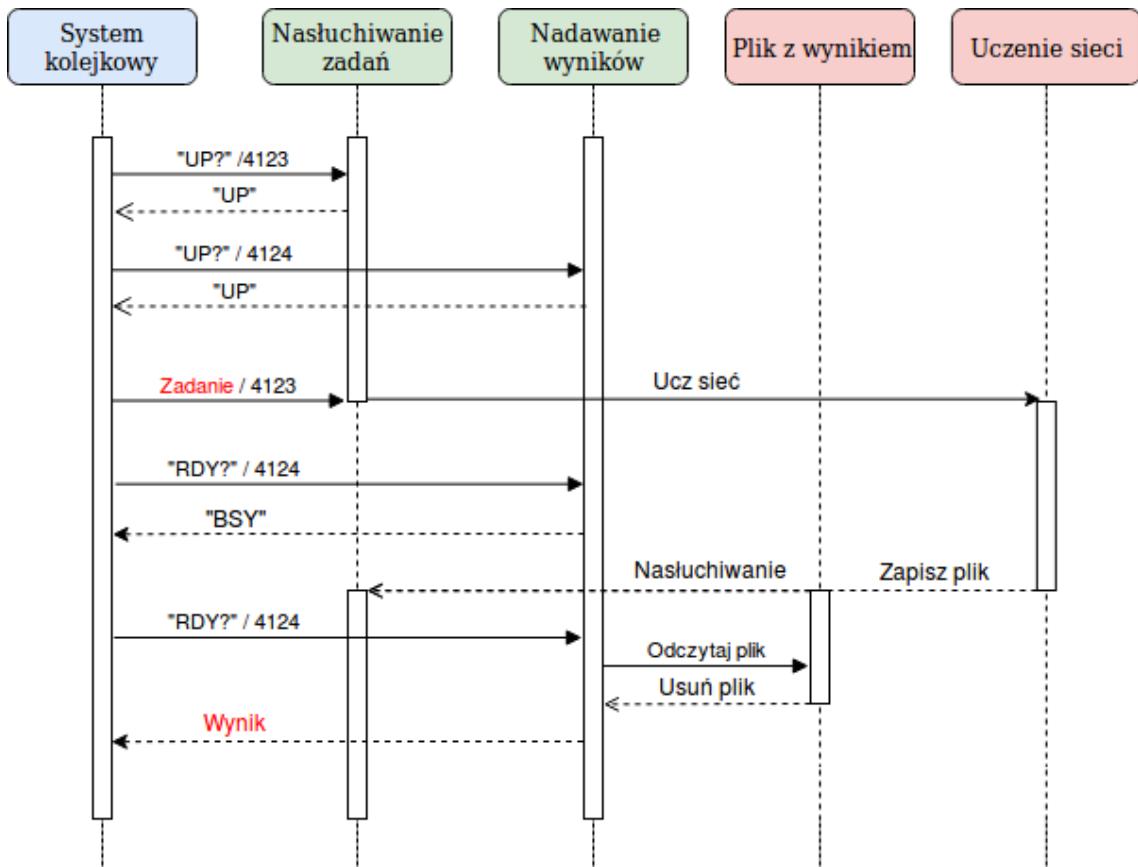
#### *Protokół komunikacji ze zdalnymi węzłami*

Komunikacja ze zdalnymi węzłami odbywa się przez protokół TCP/IP z wykorzystaniem interfejsu *socket* języka Python.

Schemat wymiany komunikatów przedstawiony został na rysunku 3.6

Widoczne u góry bloki komunikują się ze sobą zgodnie ze shematem. Pierwszy (niebieski) blok reprezentuje komunikujący się z węzłem obliczeniowym system kolejkowania zadań. Pozostałe 4 bloki stanowią podsystem pojedynczego węzła obliczeniowego. Na zielono zaznaczono obiekty komunikujące się przez sieć, natomiast na czerwono zaznaczono bloki niedostępne bezpośrednio z poziomu węzła zarządzającego.

Sama komunikacja przebiega w następujący sposób.



Rys. 3.6. Schemat wymiany komunikatów pomiędzy systemem kolejkowym a węzłami obliczeniowymi  
 Źródło: praca własna

1. System kolejkowy sprawdza, czy pod zadanym adresem IP rzeczywiście znajduje się węzeł obliczeniowy. Sprawdzenie to polega na przesłaniu tekstowo zapytania "UP?" na obydwa otwarte porty węzła obliczeniowego. Jeżeli w odpowiedzi z obydwu portu dostaje komunikat "UP", dodaje węzeł do swojej wewnętrznej listy węzłów i nie powtarza więcej tej komunikacji do momentu, kiedy znowu będzie potrzeba dodania tego węzła do listy.
2. Jeżeli poprzedni punkt przebiegł pomyślnie, węzeł jest gotowy do przyjęcia zadania. System kolejkowy przesyła spakowany przy pomocy modułu *Pickle* obiekt klasy *Job* na port 4123, który jest używany do akceptacji zadań.
3. Następnie cyklicznie (co sekundę, jest to wspominane w 3.4.3 opóźnienie przesyłane jest zapytanie na port 4124 czy jest już wynik poprzez komunikat "RDY?". Jeżeli wynik jeszcze nie jest dostępny, odsyłana jest odpowiedź "BSY" oznaczająca dalsze trwanie obliczeń. W przeciwnym wypadku przesyłany jest spakowany przy pomocy modułu *Pickle* obiekt klasy *Individual* zawierający w sobie wyniki obliczeń.
4. System kolejkowy następnie powtarza kroki 2 i 3 tak długo, jak posiada zadania do rozdzielenia, lub dopóki nie nastąpi nieoczekiwany błąd.

#### Węzeł obliczeniowy

Przez węzeł obliczeniowy rozumieć będziemy komputer lub maszynę wirtualną z dostępem do sieci komputerowej, w której znajduje się węzeł zarządzający. Dodatkowym wymaganiem na taki węzeł jest możliwość uruchomienia na nim skryptów *listener\_results.py* oraz *listener\_jobs.py*, przy czym ten drugi powinien być cyklicznie uruchamiany w przypadku wystąpienia błędu. Implikuje to obecność na danej maszynie interpretera języka Python w wersji 3 wraz z odpowiednimi bibliotekami, tj. *TensorFlow*[1], *Keras*[7] oraz *Numpy*[3]. Co więcej, urządzenie to musi mieć możliwość przyjmowania połączeń od systemu kolejkowego na portach TCP 4123 oraz TCP 4124.

Zadaniem węzła obliczeniowego jest przyjęcie zadania od systemu kolejkowego, wykonanie go i dostarczenie wyniku, również na żądanie systemu kolejkowego, przy spełnianiu wyspecyfikowanego w 3.4.4 protokołu komunikacyjnego. W związku z tym kod uruchamiany na węźle został rodzielony na 3 pliki źródłowe, każdy odpowiedzialny za jedno z tych zadań.

Pierwszym z nich jest *listener\_jobs.py*. Działa on według następującego schematu:

1. Ustawia socket nasłuchujący połączeń na porcie TCP/4123
2. W nieskończonej pętli:
  2. 1. Akceptuje przychodzące połączenie i odbiera dane
  2. 2. Jeżeli w przychodzących danych występuje komunikat "UP?" odsyła komunikat "UP", zamknięta połączenie i pomija resztę kroków w tym przebiegu pętli.
  2. 3. Zamknie połączenie od systemu kolejkowego.
  2. 4. Rozpakowuje przychodzące dane i sprawdza czy otrzymano obiekt klasy *Job*, w przeciwnym wypadku generuje wyjątek i zakończy działanie programu.
  2. 5. Przekazuje sterowanie do funkcji *eval\_network(...)* z parametrami odczytanymi z otrzymanego obiektu.

Na czas uczenia sieci proces nie nasłuchuje połączeń - przyjęto założenie projektowe, że to system zarządzający kolejką przechowuje informację, że ten węzeł właśnie pracuje i zostanie zwolniony po odebraniu wyników.

Kolejny plik, tj. *evan\_cnn.py*, zawiera jedną metodę, która oblicza funkcję przystosowania osobnika, poprzez skonstруowanie modelu sieci, uczenie go zbiorem uczącym oraz walidację na zbiorze testowym obrazków ze zbioru CIFAR-10. Bardziej szczegółowo:

1. Ustawia zadane ziarno generatora losowego. Przyczyną dla których to robi, opisano w 3.5.
2. Ładuje do pamięci zbiór danych CIFAR-10. Jeżeli skrypt jest uruchamiany pierwszy raz, pobiera zbiór z internetu.
3. Buduje model sieci neuronowej taki jak przedstawiono na rys. 3.1. Budowanie modelu jest uniwersalne dla różnych długości fenotypu, tj. różnej zadanej ilości warstw konwolucyjnych. Warstwa agregująca zostanie wstawiona w połowie, czyli dla 4 warstw za 2. warstwą.
4. Uczy uzyskany model korzystając z optymalizatora *RMSProp* przez zadaną liczbę epok. Otrzymującą historię uczenia przechowuje w odpowiednim polu obiektu klasy *Individual*.
5. Sprawdza skuteczność uzyskanej nauczonej sieci na zbiorze testowym, j.w. zapisując uzyskaną skuteczność i wartość funkcji kosztu w odpowiednich polach.
6. Pakuje uzupełniony przez powyższe operacje obiekt klasy *Individual* przy pomocy modułu *pickle* i zapisuje wynik w pliku "result".

Przy przeprowadzaniu eksperymentów zaobserwowano niedeterministyczne występowanie wyjątku *MemoryError* oznaczającego zużycie całej dostępnej pamięci i brak miejsca na zaalokowanie nowej, powodujący zakończenie działania programu. Prawdopodobną przyczyną jest błąd programistyczny w jednej z użytych bibliotek. Zastosowanym sposobem obejścia tego problemu jest ponowne uruchomienie skryptu przez inny skrypt.

Trzeci plik *listener\_results.py* odpowiedzialny jest za przesyłanie wyników na żądanie systemu kolejkowego lub informowanie, że nie są one jeszcze dostępne. Program ten realizuje następujące kroki:

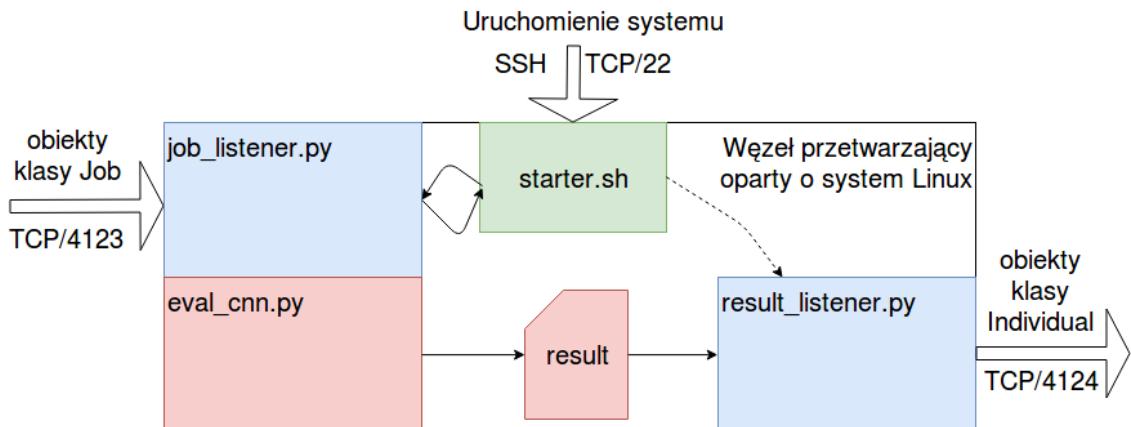
1. Ustawia socket nasłuchujący połączeń na porcie TCP/4124
2. W nieskończonej pętli:
  2. 1. Akceptuje przychodzące połączenie i odbiera dane
  2. 2. Jeżeli w przychodzących danych występuje komunikat "UP?" odsyła komunikat "UP". Jeżeli warunek nie jest spełniony, sprawdzany jest warunek poniżej.
  2. 3. Jeżeli w przychodzących danych występuje komunikat "RDY?" sprawdza czy jest dostępny plik z wynikami. Jeżeli nie, przesyła w odpowiedzi komunikat "BSY". Jeżeli jest, wysyła odczytany plik z wynikami.

## 2. 4. Zamyka połączenie z systemem kolejkowym.

Do poprawnego działania węzła obliczeniowego konieczne jest współbieżne uruchomienie skryptów *listener\_results.py* oraz *listener\_jobs.py*. Co więcej, w związku ze wspomnianymi niedeterministycznymi błędami przy obliczaniu wartości funkcji przystosowania, program zbierający zadania powinien być restartowany po każdym krytycznym błędzie. W związku z wykorzystaniem maszyn wirtualnych opartych o system Linux, w celu spełnienia dwóch powyższych warunków napisany został skrypt w języku Bash o nazwie *starter.sh*. Aby węzeł obliczeniowy zaczął działać, należy uruchomić w tle ten skrypt. Jego działanie opisać można następująco:

1. Usuń plik z wynikami - zakładamy, że startujemy od nowa i nie powinno być wyników - realizowane przez komendę *rm result*
2. Wyłącz działające procesy nasłuchujące zadań i wyników - realizowane przez komendę *pkill python*
3. Uruchom w tle i uniezależnij od procesu rodzica skrypt *listener\_results.py*
4. W nieskończonej pętli:
  4. 1. Uruchom skrypt *listener\_jobs.py* i czekaj na jego wykonanie
  4. 2. Poczekaj 10 sekund - oczekiwanie przed kolejnym uruchomieniem programu w przypadku, gdyby zajęty port TCP/4123 nie został jeszcze zwolniony.

Tak złożony system spełnia warunki postawione na węzeł obliczeniowy. Schematyczną jego reprezentację przedstawiono na rys. 3.7. Każdy element wewnętrzny prostokąta reprezentuje jeden z plików w folderze *worker*. Linia przerwana pomiędzy blokami *listener\_results.py* a *starter.sh* oznacza jedno uruchomienie. Z kolei cykl pomiędzy *listener\_results.py* a *starter.sh* symbolizuje ponowne uruchamianie skryptu. Na niebiesko zaznaczano współbieżnie dzajace skrypty komunikujące się z siecią. Na czerwono zaznaczono elementy wykorzystywane wewnętrznie przez węzeł obliczeniowy. Na zielono zaznaczono element rozpoczętający działanie systemu.



Rys. 3.7. Schemat systemu obsługi zadań  
Źródło: praca własna

## Zarządzanie infrastrukturą węzłów obliczeniowych

Opisany w 3.4.5 węzłem obliczeniowym może być, jak wspomniano tamże, dowolna maszyna z systemem operacyjnym wspierającym uruchamianie równolegle procesów interpretera języka Python. Dobrym rozwiązaniem okazują się być maszyny wirtualne z systemem Linux w chmurze. Poniżej przedstawiono listę kilku dostawców takiego rozwiązania, których rozważano przy do stworzenia infrastruktury węzłów obliczeniowych:

- Amazon Web Services
- Microsoft Azure
- Google Cloud Platform

- DigitalOcean
- Oktawave
- OVH

Na dzień dzisiejszy firma *Microsoft* oferuje 100\$ kredytu do wykorzystania dla studentów w celach badawczych, dlatego rozwiązanie to zostało wykorzystane do utworzenia 20 instancji maszyn wirtualnych, które wykorzystano jako węzły obliczeniowe. Subskrypcja dla studentów ma jednak pewne ograniczenia. W trakcie prototypowego tworzenia infrastruktury napotkano ograniczenie możliwości działających vCPU w jednym regionie do 4. Tym samym nie można było korzystać z maszyn wirtualnych wyposażonych w więcej vCPU, między innymi z przedstawionych w tabeli 3.1.

**Tabela 3.1.** Maszyny wirtualne zopytmalizowane pod kątem głębokiego uczenia. Źródło: [4]

Rozmiar	vCPU	Pamięć [GiB]	Pojemność tymczasowa (SSD) [GiB]	GPU	Maksymalnie dysków z danymi	Maksymalnie kart sieciowych
Standard_ND6s	6	112	736	1	12	4
Standard_ND12s	12	224	1474	2	24	8
Standard_ND24s	24	448	2948	4	32	8

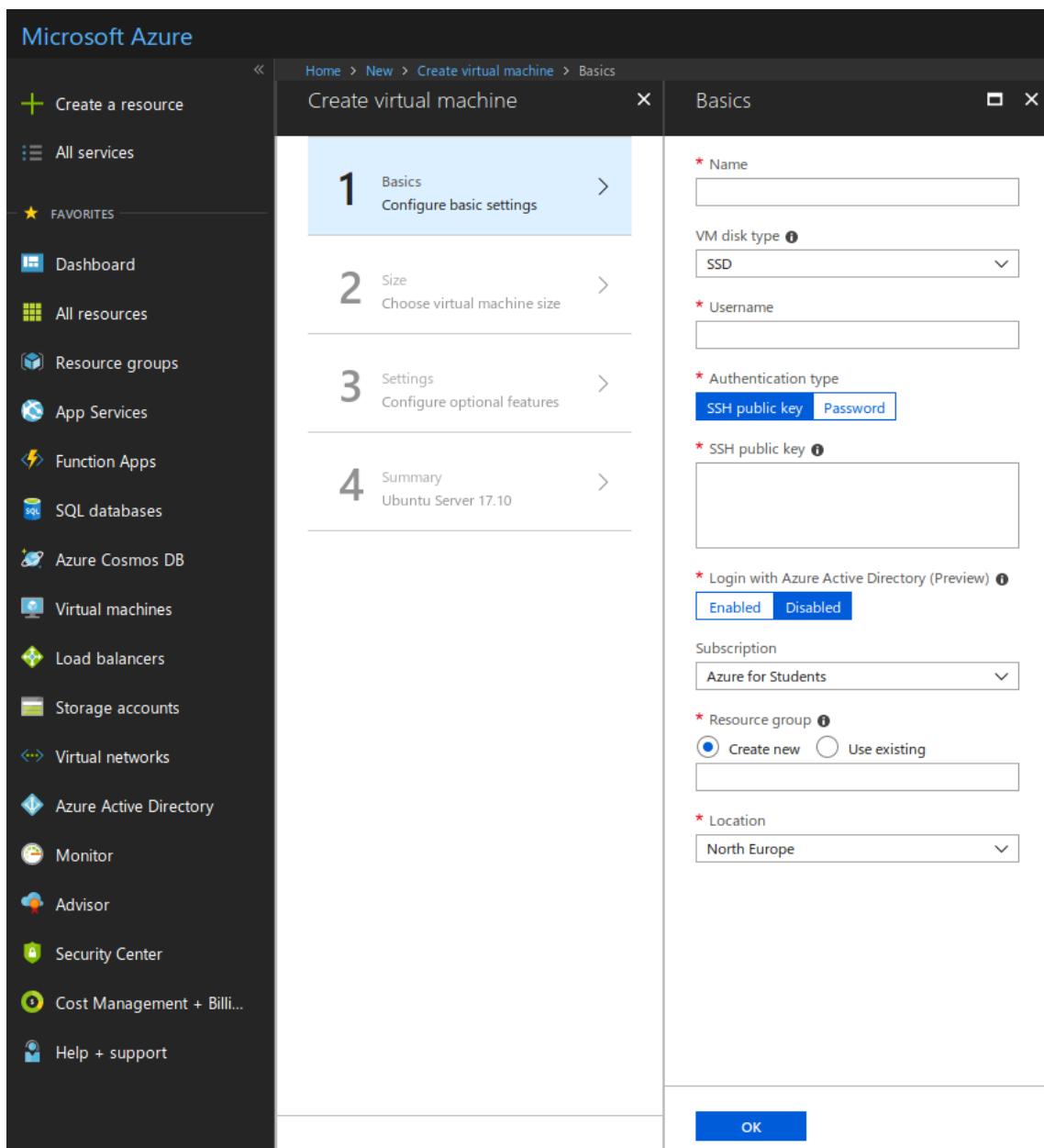
Blok zarządzania infrastrukturą węzłów obliczeniowych widoczny na rys. 3.3 po prawej stronie może reprezentować człowieka - operatora, który ręcznie utworzy wszystkie instancje korzystając z dostępnego na stronie <http://portal.azure.com> interfejsu WWW. Na rys. 3.8 przedstawiono interfejs webowy *Microsoft Azure*.

Jak widać, oprócz początkowego wybrania obrazu (tutaj wybrano Ubuntu 17.10) należy przejść przez 4 etapy, gdzie musi podjąć następujące kroki:

1. Wprowadzić nazwę maszyny wirtualnej, np. *Worker1*
2. Wprowadzić dane do logowania przez ssh: nazwę użytkownika i klucz publiczny
3. Wybrać lub stworzyć grupę zasobów (nazwano grupę *gacnn*)
4. Lokalizację - przy czym należy pamiętać o limicie 4 vCPU na lokalizację
5. Wybrać rozmiar maszyny wirtualnej - tutaj wybrano rozmiar nazwany **DS1\_v2** o następujących parametrach:
  - liczba vCPU: 1
  - Pamięć RAM: 3.5 GB
  - Rozmiar lokalnego SSD: 7 GB
  - Cena: 41.41€ za miesiąc (0.0557€ za godzinę)
6. Raz w każdym regionie stworzyć zestaw reguł zapory ogniodzielnej (*Network Security Group*) i dodać do niego regułę akceptującą połączenia na portach TCP: 4123-4124
7. Przypisać utworzony powyżej zestaw reguł do tworzonej maszyny wirtualnej
8. Wyłączyć nieporzebną opcję diagnostyki uruchamiania (*Boot Diagnostics*)
9. Zatwierdzić utworzenie maszyny wirtualnej

Rozwiążanie takie jest jednak czasochłonne i niefektywne, przejście przez wszystkie etapy przy dużym poziomie koncentracji operatora zajmuje ok. 3 minuty, co oznacza, że utworzenie 20 maszyn wirtualnych zajęłoby około godziny.

Rozwiązaniem tego problemu jest wykrzostanie narzędzia *Azure CLI*, czyli interfejs linni komend dla Azure. Pozwala on na zarządzanie maszynami wirtualnymi przy pomocy komend wydawanych z linii poleceń lub z poziomu skryptów napisanych w języku *Bash* lub *PowerShell*. Zostały napisane dwa skrypty w języku bash. Pierwszy to *spawner.sh*, który tworzy 20 maszyn wirtualnych w 5 regionach po 4 maszyny za pomocą następującej komendy wywoływanej w pętli:



Rys. 3.8. Interfejs Microsoft Azure - tworzenie maszyny wirtualnej  
Źródło: praca własna

```
az vm create --location $region --name $name
--resource-group gacnn --size Standard_DS1_v2
--image Canonical:UbuntuServer:17.10:latest
--admin-username admin
--ssh-key-value id_rsa.pub
--no-wait
```

Zmienna *region* przyjmuje jedną z wartości: *northeurope*, *westeurope*, *francecentral*, *eastus*, *eastus2*.

Zmienna *name* tworzona jest w następujący sposób:

```
name=$region" worker" $i
```

Dzięki tej tworzone jest 20 maszyn o rozmiarze **DS1\_v2** o unikalnych nazwach. Każda przypisywana jest do grupy zasobów *gacnn*. Wytwarzane są na podstawie najnowszego dostępnego obrazu *Ubuntu Server 17*. Logowanie do takiej maszyny możliwe jest z loginem *admin* i wyłącznie przy pomocy klucza publicznego z pliku *id\_rsa.pub*.

Po utworzeniu maszyn należy im również otworzyć odpowiednie porty, odbywa się to przy pomocy komendy wywoływanej w pętli:

```
az vm open-port --port 4123-4124 --resource-group gacnn --name $name
```

Z tak przygotowanymi maszynami można się połączyć przy pomocy programu, który pozowała obsługiwać wiele terminali naraz, np. *Cluster SSH*. Adresy IP maszyn odczytać można korzystając już z interfejsu WWW Azure, wyświetlając wszystkie dostępne maszyny wirtualne i ich publiczne adresy IP. Poniższe kroki można w ten sposób wykonywać na 20 maszynach jednocześnie.

Dalsze przygotowanie środowiska dla węzła obliczeniowego polega na:

1. pobraniu programu *python3-pip*
2. instalacja za jego pomocą bibliotek *numpy*, *tensorflow*, *keras*.
3. sklonowanie repozytorium umieszczonego pod adresem <https://github.com/opiechow/gacnn>
4. przejście do katalogu *gacnn/worker*
5. uruchomienie skrytu *starer.sh*

Po tych operacjach węzły obliczeniowe działają i czekają na zadania od systemu kolejkowego. Można zmodyfikować plik *starter.sh*, aby przekierować wyjście ze skryptu *listener\_jobs.py* na standarowe wyjście, co pozwoli obserwować proces uczenia w terminalu. Przykładowy widok dla sześciu węzłów obliczeniowych przedstawiono na rys. 3.9

```

CSSH: root@167.99.251.110 - + ×
42336/50000 [=====>.....] - ETA: 94s - loss: 1.8739 - acc: 0, 4864/50000 [=>.....] - ETA: 409s - loss: 2.2382 - acc: 0
42368/50000 [=====>.....] - ETA: 94s - loss: 1.8736 - acc: 0, 4996/50000 [=>.....] - ETA: 408s - loss: 2.2375 - acc: 0
42400/50000 [=====>.....] - ETA: 93s - loss: 1.8733 - acc: 0, 4928/50000 [=>.....] - ETA: 408s - loss: 2.2362 - acc: 0
42432/50000 [=====>.....] - ETA: 93s - loss: 1.8730 - acc: 0, 4960/50000 [=>.....] - ETA: 408s - loss: 2.2349 - acc: 0
42464/50000 [=====>.....] - ETA: 93s - loss: 1.8727 - acc: 0, 4992/50000 [=>.....] - ETA: 408s - loss: 2.2338 - acc: 0
42496/50000 [=====>.....] - ETA: 92s - loss: 1.8726 - acc: 0, 5024/50000 [=>.....] - ETA: 407s - loss: 2.2330 - acc: 0
42528/50000 [=====>.....] - ETA: 92s - loss: 1.8725 - acc: 0, 5056/50000 [=>.....] - ETA: 407s - loss: 2.2317 - acc: 0
42560/50000 [=====>.....] - ETA: 91s - loss: 1.8725 - acc: 0, 5120/50000 [=>.....] - ETA: 406s - loss: 2.2304 - acc: 0
42592/50000 [=====>.....] - ETA: 91s - loss: 1.8726 - acc: 0, 5120/50000 [=>.....] - ETA: 406s - loss: 2.2301 - acc: 0
42624/50000 [=====>.....] - ETA: 91s - loss: 1.8725 - acc: 0, 5184/50000 [=>.....] - ETA: 406s - loss: 2.2291 - acc: 0
42656/50000 [=====>.....] - ETA: 90s - loss: 1.8722 - acc: 0, 5184/50000 [=>.....] - ETA: 405s - loss: 2.2288 - acc: 0
42688/50000 [=====>.....] - ETA: 90s - loss: 1.8720 - acc: 0, 5248/50000 [=>.....] - ETA: 405s - loss: 2.2277 - acc: 0
42720/50000 [=====>.....] - ETA: 89s - loss: 1.8718 - acc: 0, 5280/50000 [=>.....] - ETA: 405s - loss: 2.2272 - acc: 0
42752/50000 [=====>.....] - ETA: 89s - loss: 1.8718 - acc: 0, 5312/50000 [=>.....] - ETA: 404s - loss: 2.2257 - acc: 0
42784/50000 [=====>.....] - ETA: 89s - loss: 1.8716 - acc: 0, 5344/50000 [=>.....] - ETA: 404s - loss: 2.2244 - acc: 0
42816/50000 [=====>.....] - ETA: 88s - loss: 1.8715 - acc: 0, 5376/50000 [=>.....] - ETA: 404s - loss: 2.2232 - acc: 0
3328[]

CSSH: root@167.99.251.135 - + ×
4864/50000 [=>.....] - ETA: 409s - loss: 2.2382 - acc: 0
4996/50000 [=>.....] - ETA: 408s - loss: 2.2375 - acc: 0
4928/50000 [=>.....] - ETA: 408s - loss: 2.2362 - acc: 0
4960/50000 [=>.....] - ETA: 408s - loss: 2.2349 - acc: 0
4992/50000 [=>.....] - ETA: 408s - loss: 2.2338 - acc: 0
5024/50000 [=>.....] - ETA: 407s - loss: 2.2330 - acc: 0
5056/50000 [=>.....] - ETA: 407s - loss: 2.2317 - acc: 0
5120/50000 [=>.....] - ETA: 406s - loss: 2.2304 - acc: 0
5120/50000 [=>.....] - ETA: 406s - loss: 2.2301 - acc: 0
5184/50000 [=>.....] - ETA: 406s - loss: 2.2291 - acc: 0
5184/50000 [=>.....] - ETA: 405s - loss: 2.2288 - acc: 0
5248/50000 [=>.....] - ETA: 405s - loss: 2.2277 - acc: 0
5280/50000 [=>.....] - ETA: 405s - loss: 2.2272 - acc: 0
5312/50000 [=>.....] - ETA: 404s - loss: 2.2257 - acc: 0
5344/50000 [=>.....] - ETA: 404s - loss: 2.2244 - acc: 0
5376/50000 [=>.....] - ETA: 404s - loss: 2.2232 - acc: 0
1760[]

CSSH: root@167.99.251.89 - + ×
9632/50000 [====>.....] - ETA: 625s - loss: 2.2044 - acc: 0, 12928/50000 [====>.....] - ETA: 529s - loss: 2.0490 - acc: 0
9864/50000 [====>.....] - ETA: 625s - loss: 2.2038 - acc: 0, 12960/50000 [====>.....] - ETA: 529s - loss: 2.0483 - acc: 0
9896/50000 [====>.....] - ETA: 624s - loss: 2.2038 - acc: 0, 12992/50000 [====>.....] - ETA: 528s - loss: 2.0479 - acc: 0
9728/50000 [====>.....] - ETA: 624s - loss: 2.2036 - acc: 0, 13024/50000 [====>.....] - ETA: 528s - loss: 2.0479 - acc: 0
9760/50000 [====>.....] - ETA: 624s - loss: 2.2034 - acc: 0, 13056/50000 [====>.....] - ETA: 527s - loss: 2.0475 - acc: 0
9792/50000 [====>.....] - ETA: 623s - loss: 2.2031 - acc: 0, 13088/50000 [====>.....] - ETA: 527s - loss: 2.0469 - acc: 0
9824/50000 [====>.....] - ETA: 622s - loss: 2.2027 - acc: 0, 13120/50000 [====>.....] - ETA: 526s - loss: 2.0461 - acc: 0
9856/50000 [====>.....] - ETA: 622s - loss: 2.2021 - acc: 0, 13152/50000 [====>.....] - ETA: 526s - loss: 2.0455 - acc: 0
9888/50000 [====>.....] - ETA: 621s - loss: 2.2018 - acc: 0, 13184/50000 [====>.....] - ETA: 525s - loss: 2.0452 - acc: 0
9920/50000 [====>.....] - ETA: 620s - loss: 2.2014 - acc: 0, 13216/50000 [====>.....] - ETA: 525s - loss: 2.0444 - acc: 0
9952/50000 [====>.....] - ETA: 620s - loss: 2.2006 - acc: 0, 13248/50000 [====>.....] - ETA: 525s - loss: 2.0442 - acc: 0
9984/50000 [====>.....] - ETA: 619s - loss: 2.2003 - acc: 0, 13280/50000 [====>.....] - ETA: 524s - loss: 2.0436 - acc: 0
10016/50000 [====>.....] - ETA: 619s - loss: 2.1998 - acc: 0, 13312/50000 [====>.....] - ETA: 524s - loss: 2.0430 - acc: 0
10048/50000 [====>.....] - ETA: 618s - loss: 2.1994 - acc: 0, 13344/50000 [====>.....] - ETA: 523s - loss: 2.0428 - acc: 0
10080/50000 [====>.....] - ETA: 618s - loss: 2.1984 - acc: 0, 13376/50000 [====>.....] - ETA: 523s - loss: 2.0424 - acc: 0
10112/50000 [====>.....] - ETA: 617s - loss: 2.1982 - acc: 0, 13408/50000 [====>.....] - ETA: 522s - loss: 2.0419 - acc: 0
10144/50000 [====>.....] - ETA: 617s - loss: 2.1980 - acc: 0, 13440/50000 [====>.....] - ETA: 522s - loss: 2.0420 - acc: 0
1896[]

CSSH: root@167.99.251.94 - + ×
5568/50000 [==>.....] - ETA: 520s - loss: 2.2180 - acc: 0
5600/50000 [==>.....] - ETA: 520s - loss: 2.2166 - acc: 0
5632/50000 [==>.....] - ETA: 521s - loss: 2.2153 - acc: 0
5664/50000 [==>.....] - ETA: 521s - loss: 2.2142 - acc: 0
5696/50000 [==>.....] - ETA: 521s - loss: 2.2129 - acc: 0
5728/50000 [==>.....] - ETA: 522s - loss: 2.2121 - acc: 0
5760/50000 [==>.....] - ETA: 522s - loss: 2.2112 - acc: 0
5792/50000 [==>.....] - ETA: 523s - loss: 2.2104 - acc: 0
5824/50000 [==>.....] - ETA: 523s - loss: 2.2098 - acc: 0
5856/50000 [==>.....] - ETA: 523s - loss: 2.2088 - acc: 0
5888/50000 [==>.....] - ETA: 524s - loss: 2.2077 - acc: 0
5920/50000 [==>.....] - ETA: 524s - loss: 2.2068 - acc: 0
5952/50000 [==>.....] - ETA: 524s - loss: 2.2061 - acc: 0
5984/50000 [==>.....] - ETA: 525s - loss: 2.2051 - acc: 0
6016/50000 [==>.....] - ETA: 525s - loss: 2.2030 - acc: 0
6048/50000 [==>.....] - ETA: 525s - loss: 2.2029 - acc: 0
6080/50000 [==>.....] - ETA: 526s - loss: 2.2017 - acc: 0
1972[]

CSSH: root@167.99.251.94 - + ×

```

Rys. 3.9. 6 węzłów obliczeniowych w trakcie pracy

Źródło: praca własna

### **Replikatywność badań**

Zarówno w przypadku działania algorytmu genetycznego, jak i w przypadku uczenia sieci neuronowych, występuje wiele operacji, które z założenia powinny być niedeterministyczne. Poniżej przedstawiono listę tych operacji dla algorytmu genetycznego:

- Losowe generowanie początkowej populacji algorytmu genetycznego
- Selekcja ruletkowa
- Losowe dobieranie osobników w pary rodzicielskie
- Zachodzenie krzyżowania z pewnym prawdopodobieństwem
- Losowanie maski w krzyżowaniu równomiernym
- Zachodzenie mutacji z pewnym prawdopodobieństwem
- Losowanie parametru do zmutowaniu
- Losowanie nowej wartości mutowanego parametru
- Losowanie części populacji przechodzącej do kolejnej iteracji

W przypadku uczenia sieci neuronowych losowo inicjalizowane są początkowe wagi w sieci.

Aby zapewnić replikatywność badań, w toku pisania systemu dbano o zachowanie kontroli nad stanem początkowym generatora liczb pseudolosowych. W tym celu zarówno przed uruchomieniem algorytmu genetycznego jak i uczeniem sieci ustawiane jest ziarno generatora losowego. W przypadku węzła zarządzającego ziarno jest jednym z parametrów ustawianym przed uruchomieniem algorytmu. W przypadku węzłów obliczeniowych ziarno dla generatora odbierane jest razem z zadaniem do policzenia.

W przypadku algorytmu genetycznego używany jest generator z wbudowanego w język *Python* modułu *random*, ustawienie ziarna nie jest skomplikowane - wystarczy jedna linia kodu. W przypadku uczenia sieci neuronowych wszystkie operacje losowe ukryte są w działaniu funkcji biblioteki *Keras*. Procedura ustawiania ziarna dla wszystkich generatorów dostępna jest w dokumentacji biblioteki. [7]

Poważną wadą dbania o replikatywność jest wymuszenie na *TensorFlow* działania wyłącznie na jednym wątku, gdyż uniemożliwia to korzystanie z korzyści oferowanych przez uczenie sieci przy pomocy GPU i technologii CUDA. Aby umożliwić działanie systemu w tych technologiami, kryterium stopu uzależniono od wariancji wyników uzyskanych dla takiej samej sieci korzystając z różnych wartości ziarna generatora liczb losowych.

W przypadku przedstawionych poniżej eksperymentów nie korzystano z uczenia GPU, a z 20 maszyn wirtualnych z jednym vCPU każda. Ziarno ustawiane było wg. opisanej powyżej procedury, zatem wszystkie badania powinny być reprodukowalne.

## Badania i testy

### Wariancja wyników dla jednej sieci

Ważnym parametrem algorytmu genetycznego jest jego kryterium stopu. W przypadku eksperymentów opisanych w 4.2 i 4.4 przyjęto dwa:

1. maksymalna liczba iteracji - z przyczyn długiego czasu potrzebnego na obliczenia w przypadku eksperymentu 4.4, liczbę tę ustalono na 100.
2. maksymalna liczba iteracji bez poprawy - jeżeli od dłuższego czasu nie obserwujemy poprawy rozwiązania, możemy przyjąć założenie, że znaleźliśmy globalne minimum

Z punktem 2 wiąże się parametr  $\epsilon$ , który ma sens progu różnicy między nową najlepszą wartością a poprzednią, po przekroczeniu którego możemy mówić o poprawie.

Program eksperymentu został tak zaimplementowany, by reprodukowalność eksperymentu była możliwie najwyższa, tj. by niwelować wpływ generatora pseudolosowego na uzyskane wyniki. W procesie uczenia sieci neuronowych losowo inicjalizowane są wagi połączeń między neuronami. Oznacza to, że w przypadku, gdy nie mamy kontroli nad stanem generatora losowego, ta sama w sensie struktury sieć może uzyskać inny wynik po jednej epoce uczenia w zależności od początkowych wag.

W celu minimalizacji wpływu losowego początkowego doboru wag na poprawę wyników sieci, przeprowadzono 100-krotnie uczenie sieci o 4 warstwach, po jednym filtrze w każdej, z różnymi początkowymi wagami.

Następnie wyznaczono wartość średnią (wzór 4.1) wariancję (wzór 4.2) zebranych pomiarów.

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad (4.1)$$

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2 \quad (4.2)$$

Dla 100 prób wariancja wyniosła  $\sigma^2 = 3.708 \times 10^{-6}$ , a odchylenie standardowe (pierwiastek kwadratowy z wariancji)  $\sigma = 1.9255 \times 10^{-3}$ . W związku z tym przyjęto wartość parametru  $\epsilon = 4\sigma = 7.702 \times 10^{-3}$ , aby wpływ różnej inicjalizacji wag nie był interpretowany przez algorytm genetyczny jako poprawa.

### Test działania algorytmu genetycznego

W celu zbadania poprawności działania oraz dostrojenia parametrów algorytmu genetycznego przeprowadzono test szukania minimum funkcji Levego. Funkcja Levego (*Levy Function*) dana jest wzorem 4.3:

$$f(\mathbf{x}) = \sin^2(\pi w_1) + \sum_{i=1}^{d-1} (w_i - 1)^2 [1 + 10 \sin^2(\pi w_i + 1)] + (w_d - 1)^2 [1 + \sin^2(2\pi w_d)] \quad (4.3)$$

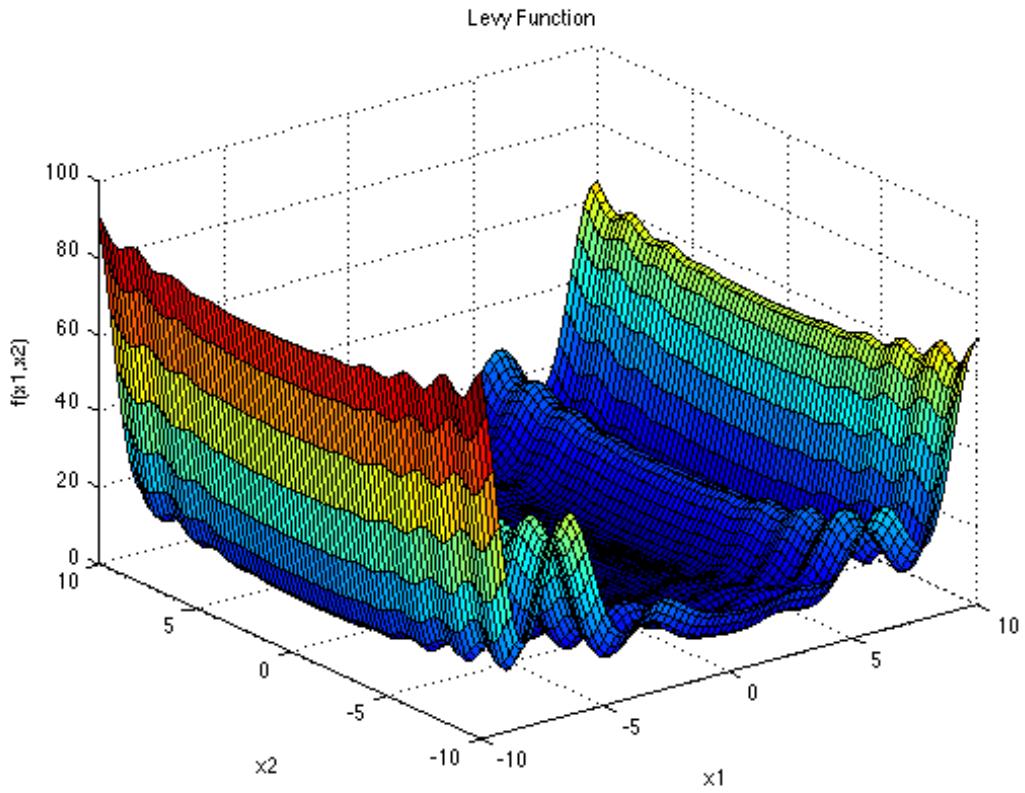
gdzie  $w_i$  obliczane jest ze wzoru 4.4, natomiast  $d$  oznacza liczbę wymiarów.

$$w_i = 1 + \frac{x_i - 1}{4} \quad (4.4)$$

Minimum globalne tej funkcji znajduje się w punkcie  $\mathbf{x}^* = (1, \dots, 1)$  a jego wartość wynosi zero, tj.  $f(\mathbf{x}^*) = 0$ . [18] Wykres tej funkcji dla przypadku 2-wymiarowego przedstawiono na rys. 4.1

Przestrzeń poszukiwań (genotyp osobnika) przyjęto identyczną jak dla problemu optymalizacji struktury konwonlucyjnych sieci neuronowych, tj.  $x_i \in \{1, 2, \dots, 30, 31\}$  dla  $i = 1..4$ .

W celu uniknięcia przypadku, w którym optymalny osobnik powstaje w wyniku mechanizmu naprawiania osobników (opisanego w 3.3.4) dokonano arbitralnie dobranego przesunięcia wektora  $\mathbf{x}$  o



Rys. 4.1. Wykres funkcji Levego

Źródło: [18]

wektor  $\begin{bmatrix} 3 \\ 7 \\ 13 \\ 20 \end{bmatrix}$ , tak, że optymalne rozwiązanie znalazło się w punkcie  $\mathbf{x}^* = \begin{bmatrix} 4 \\ 8 \\ 14 \\ 21 \end{bmatrix}$ .

Jako, że algorytm maksymalizuje funkcję przystosowania osobników, w celu minimalizacji wartości funkcji Levego (funkcji kosztu) została ona zdefiniowana zgodnie ze wzorem 4.5. [5]

$$g(\mathbf{x}) = C_{max} - f(\mathbf{x}) \quad (4.5)$$

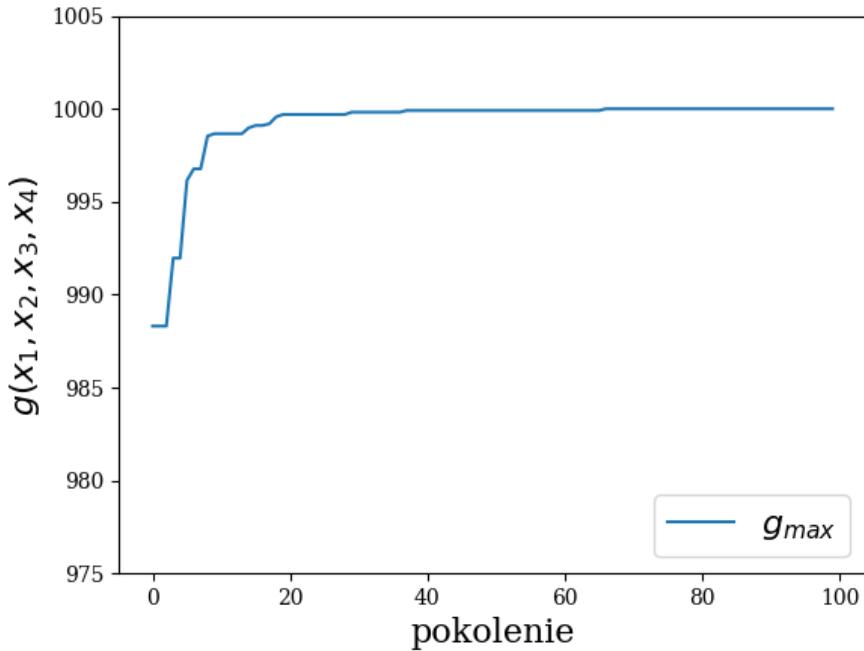
gdzie  $C_{max} = 1000$  w celu spełnienia warunku  $g(\mathbf{x}) > 0$  dla każdego  $\mathbf{x}$  w przestrzeni poszukiwań.

Metodą prób i błędów dobrano takie nastawy parametrów algorytmu genetycznego, by znajdował on minimum tej funkcji, tj.

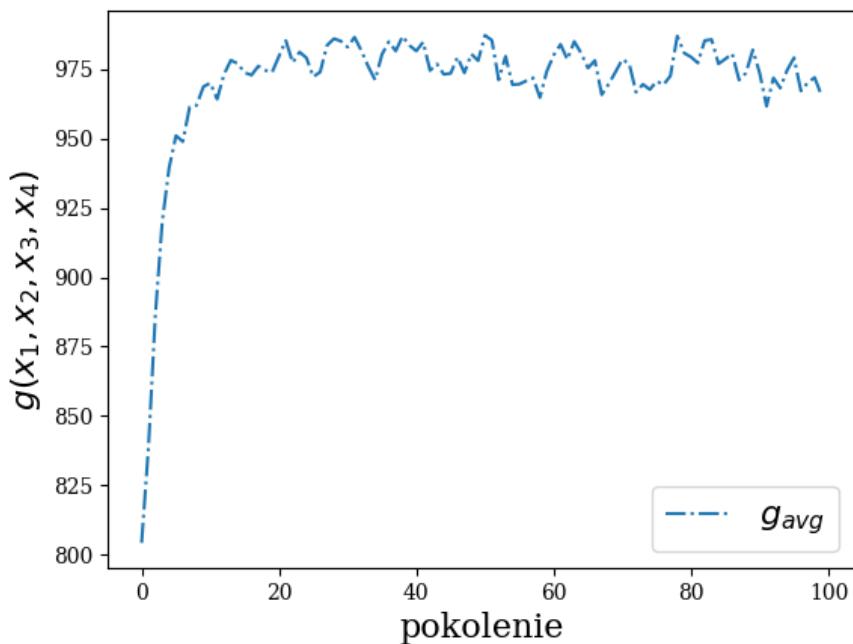
- Liczba osobników  $N = 100$
- prawdopodobieństwo krzyżowania -  $p_c = 0.9$
- prawdopodobieństwo mutacji -  $p_m = 0.3$
- maksymalna liczba iteracji - 100
- maksymalna liczba iteracji bez poprawy - 50
- liczba najlepszych osobników przechodzących do kolejnego pokolenia -  $N_k = 30$
- minimalna poprawa  $\epsilon = 0.007702$

Ostatni parametr dobrany został w sposób opisany w 4.1 Operacje genetyczne przeprowadzane są tak jak opisano w 3.3. Fenotyp pojedynczego osobnika jest 4-elementową listą liczb całkowitych:  $[x_1, x_2, x_3, x_4]$ , natomiast genotyp jest binarną reprezentacją fenotypu na liczbach o długości  $m = 5$  bitów. Badanie zostało przeprowadzone dla ziarna generatora ustawionego na wartość 1337.

Na rys. 4.2 i rys. 4.3 przedstawiono odpowiednio maksymalne i średnie (dla całej populacji) wartości funkcji przystosowania w kolejnych iteracjach algorytmu. Wszystkie kolejne wykresy zostały wykonane przy pomocy biblioteki *Matplotlib*. [8]



Rys. 4.2. Maksymalne wartości funkcji przystosowania w każdej iteracji dla funkcji Levego  
Źródło: praca własna



Rys. 4.3. Średnie wartości funkcji przystosowania w każdej iteracji dla funkcji Levego  
Źródło: praca własna

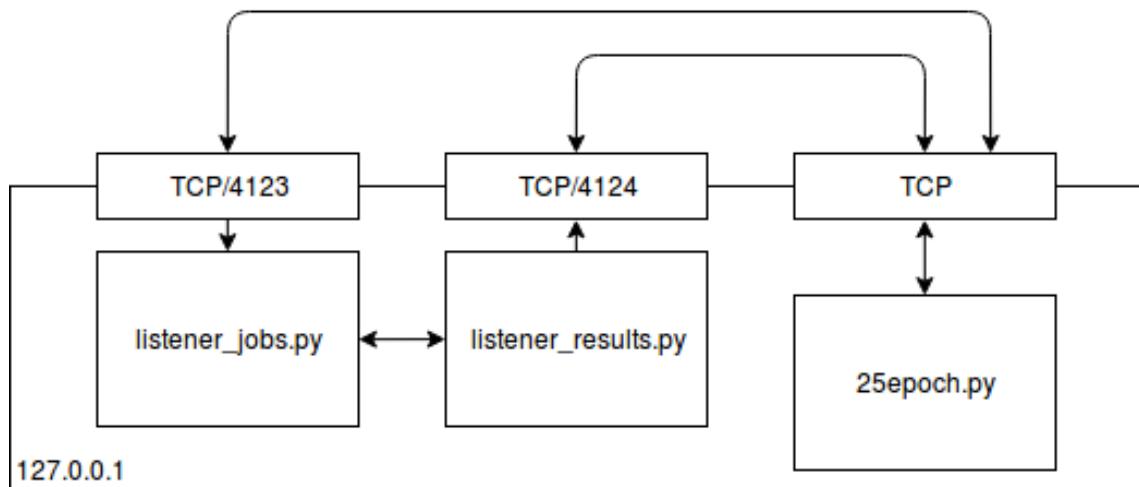
Zaobserwowane przebiegi wyglądają jak poprawne przebiegi dla algorytmu genetycznego. Mi-

nimum w punkcie  $\mathbf{x}^* = \begin{bmatrix} 4 \\ 8 \\ 14 \\ 21 \end{bmatrix}$  zostało znalezione w 67. pokoleniu, a kryterium zatrzymania algorytmu, które zadziałało, to maksymalna liczba iteracji równa 100.

Test ten udowodnił, że algorytm genetyczny został zaimplementowany poprawnie i w związku z tym można go w takiej formie wykorzystać do poszukiwania optymalnej struktury konwolucyjnych sieci neuronowych.

### **Test działania dla lokalnego węzła obliczeniowego**

Schemat działania testu przedstawiono na rys. 4.4



Rys. 4.4. Test działania całego systemu w ramach maszyny lokalnej

Źródło: praca własna

Kolejność wykonywania działań w teście była następująca:

1. Modyfikacja pliku *workers* by zawierał wyłącznie adres 127.0.0.1, czyli adres maszyny lokalnej
2. Uruchomienie skryptu *listener\_jobs.py* do nasłuchiwanego zadań
3. Uruchomienie skryptu *listener\_results.py* do nasłuchiwanego zapytań o wyniki
4. Uruchomienie skryptu *25epoch.py*, który zleca zadanie uczenia jednej sieci przez 25 epok

Test ten pozwolił na zweryfikowanie:

1. Czy rzeczywiście program węzła zdalnego działa niezależnie od systemu operacyjnego - na maszynie lokalnej działał system Linux Mint 18.3 Cinnamon 64 bit
2. Czy zdefiniowano wystarczająco duży bufor dla przesyłu informacji pomiędzy węzłami
3. Czy system kolejkowy działa jak powinien

W wyniku działania testu zauważono, że bufor o rozmiarze 1 kB (1024 bajtów) nie był wystarczająco duży, gdyż przesyłany plik z historią uczenia dla 25 epok miał rozmiar 1,5 kB, co powodowało ucięcie transmisji i błąd w dekodowaniu spakowanego obiektu. Dla zachowania pewnego zapasu bezpieczeństwa zwiększo rozmiar tego bufora do 8192 bajtów.

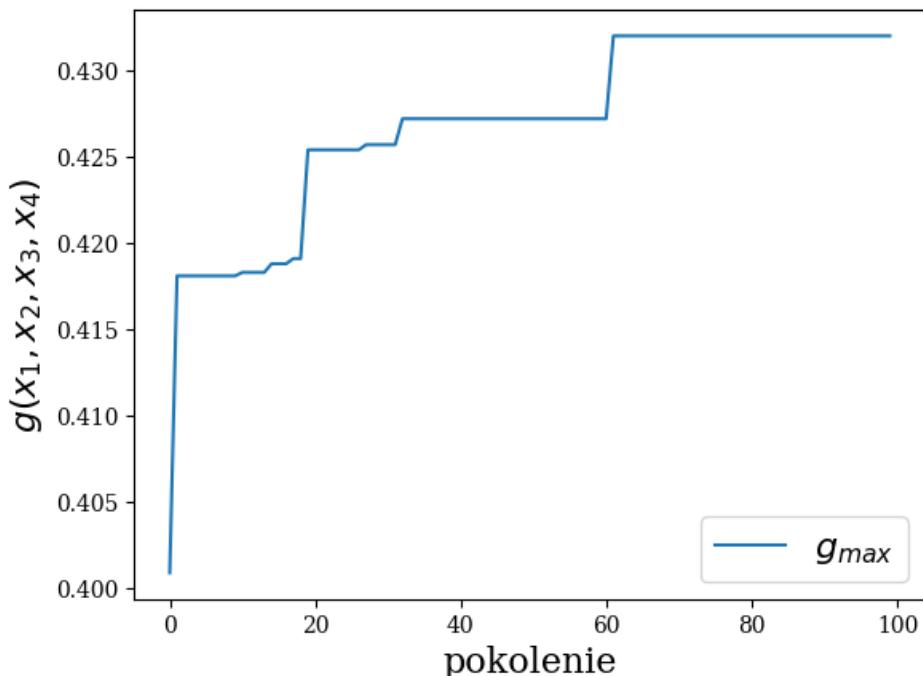
Ponadto test sprawdził działanie zarówno systemu kolejkowego dla brzegowego przypadku jednego zadania i jednego węzła, jak i oprogramowania węzła obliczeniowego. W wyniku jego przeprowadzenia otrzymano poprawnie nauczoną sieć neuronową, więcej na ten temat powiedziano w ??.

## **Poszukiwanie optymalnej struktury konwolucyjnej sieci neuronowej za pomocą algorytmu genetycznego dla uczenia 1-epokowego**

Celem tego badania jest sprawdzenie, czy algorytm genetyczny jest w stanie znaleźć optymalną w sensie skuteczności po jednej epoce uczenia strukturę sieci neuronowej.

Parametry algorytmu genetycznego przyjęto dokładnie takie, do jakich dostrojono algorytm, gdy poszukiwał minimum funkcji Levego, to jest:

- Liczba osobników  $N = 100$
- prawdopodobieństwo krzyżowania -  $p_c = 0.9$
- prawdopodobieństwo mutacji -  $p_m = 0.3$
- maksymalna liczba iteracji - 100
- maksymalna liczba iteracji bez poprawy - 50
- liczba najlepszych osobników przechodzących do kolejnego pokolenia -  $N_k = 30$
- minimalna poprawa  $\epsilon = 0.007702$



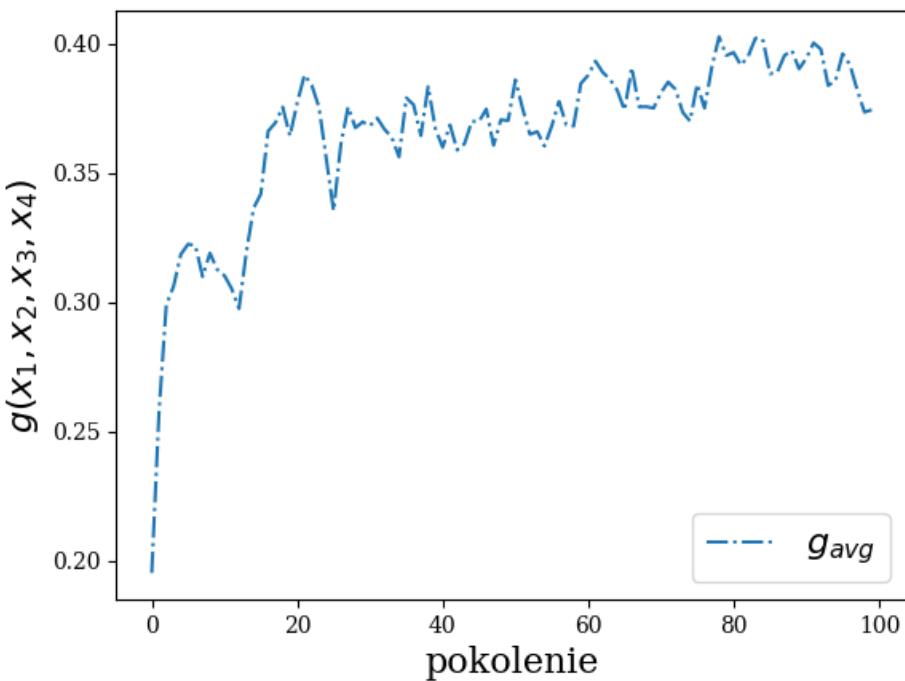
Rys. 4.5. Maksymalne wartości funkcji przystosowania w każdej iteracji dla sieci konwolucyjnych - podejście nr 1  
 Źródło: praca własna

Przeprowadzono dwa podejścia do tego eksperymentu, z powodu zaobserwowania niepokoujących zjawisk w wynikach z podejścia pierwszego. Na rys. 4.5 i rys. 4.6 obserwujemy wykresy odpowiednio najwyższych i średnich wartości funkcji przystosowania w populacji.

Badanie zostało przeprowadzone z ziarnem generatora 1337, jednak z przyczyn niezależnych algorytm przerwał działanie i był wzmnawiany odpowiednio w iteracjach 19, 27 i 54, zatem za każdym razem w tych momentach ziarno było ustawiane na 1337.

Przy pierwszym spojrzeniu nie widać nic niepokojującego. Wykresy przedstawiają poprawne działanie algorytmu genetycznego. Maksimum funkcji przystosowania wg. algorytmu znajduje się w punkcie

$$x^* = \begin{bmatrix} 4 \\ 4 \\ 13 \\ 19 \end{bmatrix} \text{ i zostało odnalezione w 62. iteracji.}$$



Rys. 4.6. Średnie wartości funkcji przystosowania w każdej iteracji dla sieci konwolucyjnych - podejście nr 1  
 Źródło: praca własna

Co widoczne będzie dopiero po porównaniu tych wykresów z wykresami na rys. 4.8 i rys. 4.9, zarówno maksymalne jak i średnie wartości utrzymują się na niższym poziomie. Jest tak ponieważ pojawiło się nadspodziewanie wiele wartości zbliżonych do 0.1, czego przyczyną może być jedno z następujących dwóch zjawisk:

1. sieć ma bardzo złą strukturę (np. 1 filtr konwolucyjny przed warstwą agregującą)
2. błędy w trakcie uczenia sieci przez węzeł obliczeniowy, które nie powodowały wyłączenia programu, jednak miały istotny wpływ na wynik

Po bliższym przyjrzeniu się kilku przypadkom z danych znaleziono przykłady, w których sieć wygląda,

jakby nie miała powodu by osiągać niskie wyniki, np.  $\mathbf{x} = \begin{bmatrix} 30 \\ 20 \\ 15 \\ 20 \end{bmatrix}$ , a jej skutczność klasyfikacji była oceniana na poziomie 10%.

Przyczyną okazało się złe skonfigurowanie węzłów obliczeniowych. Wówczas korzystano z węzłów obliczeniowych opartych na obrazie *Data Science Virtual Machine for Linux*. Obraz ten został wybrany ze względu na preinstalowane biblioteki *Keras*, *TensorFlow* i *Numpy*. Po bliższym przyjrzeniu się sytuacji i ponownym uruchomieniu konkretnego przykładu na maszynie wirtualnej *Data Science Virtual Machine for Linux* ukazało się wiele ostrzeżeń, jak na rys. 4.7.

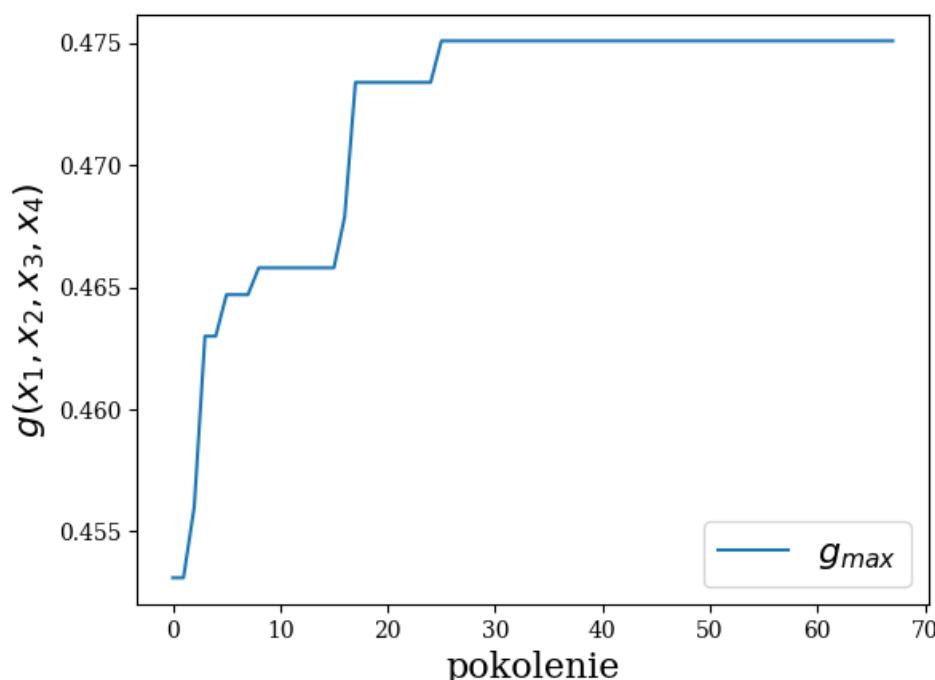
Widać, że maszyna ma problem z alokacją odpowiedniej ilości pamięci dla większych sieci, stąd niepoprawne wyniki na poziomie 10% skutczności klasyfikacji. W trakcie pierwszego podejścia przyczynę problemu odnaleziono dopiero po wykonaniu poważnych pomiarów. To podejście okazało się być podejściem testowym i pomogło dopracować system w pełni, tak by działał poprawnie. Łącznie uczenie wszystkich sieci trwało ok. 24 godzin na 20 węzłach.

Poniżej, na rys. 4.8 i rys. 4.9 przedstawiono maksymalne i średnie wartości funkcji przystosowania dla kolejnych generacji w drugim podejściu do eksperymentu.

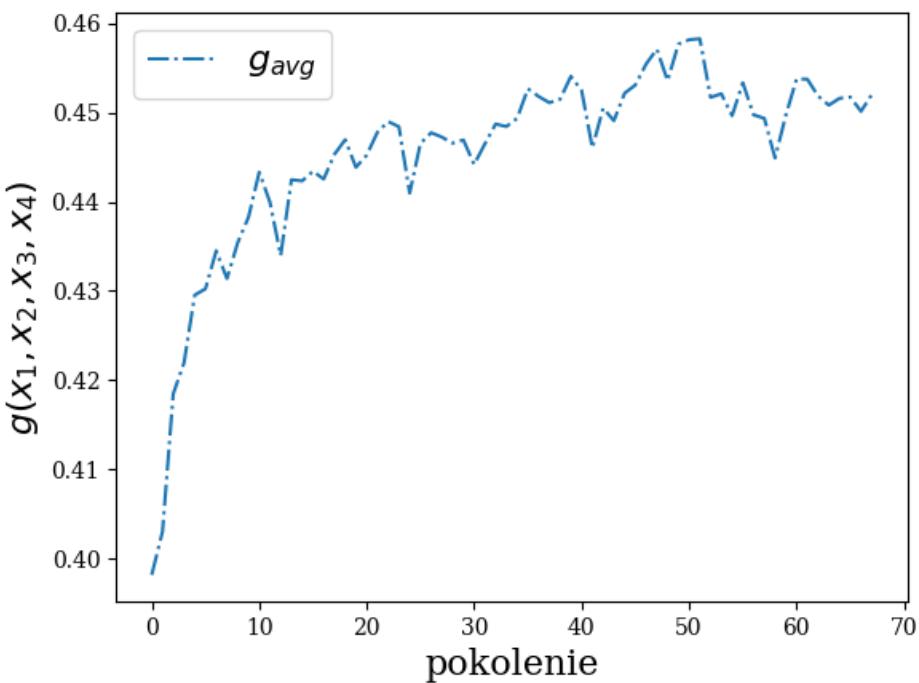
Ponownie jak w teście Levego, wykresy maksymalnych i średnich wartości funkcji przystosowania osobników wyglądają jak typowy dla poprawnie zaimplementowanego algorytmu genetycznego. Kryterium stopu które zadziałało tym razem to liczba iteracji bez poprawy - 50. Znalezione przez al-

```
valef@tester: ~/gacnn/worker
File Edit View Search Terminal Help
/anaconda/envs/py35/lib/python3.5/site-packages/h5py/_init_.py:36: FutureWarning:
Conversion of the second argument of issubdtype from `float` to `np.floating`
is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
    from ._conv import register_converters as _register_converters
Using TensorFlow backend.
2018-08-11 20:44:57.051552: I tensorflow/core/platform/cpu_feature_guard.cc:140]
Your CPU supports instructions that this TensorFlow binary was not compiled to
use: AVX2 FMA
2018-08-11 20:44:57.192098: E tensorflow/stream_executor/cuda/cuda_driver.cc:406]
failed call to cuInit: CUDA_ERROR_UNKNOWN
2018-08-11 20:44:57.192157: I tensorflow/stream_executor/cuda/cuda_diagnostics.c
c:145] kernel driver does not appear to be running on this host (tester): /proc/
driver/nvidia/version does not exist
Layers : (30, 20, 15, 20)
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [=====] - 37s 0us/step
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
2018-08-11 20:45:45.596342: W tensorflow/core/framework/allocator.cc:101] Allocat
tion of 23979780 exceeds 10% of system memory.
2018-08-11 20:45:45.612815: W tensorflow/core/framework/allocator.cc:101] Allocat
ion of 26891280 exceeds 10% of system memory.
2018-08-11 20:45:45.631059: W tensorflow/core/framework/allocator.cc:101] Allocat
ion of 26891280 exceeds 10% of system memory.
2018-08-11 20:45:45.648417: W tensorflow/core/framework/allocator.cc:101] Allocat
ion of 29160000 exceeds 10% of system memory.
2018-08-11 20:45:45.731201: W tensorflow/core/framework/allocator.cc:101] Allocat
ion of 23979780 exceeds 10% of system memory.
32/50000 [.....] - ETA: 33:32 - loss: 2.3026 - acc:
0.15622018-08-11 20:45:45.830660: W tensorflow/core/framework/allocator.cc:101]
```

Rys. 4.7. Ostrzeżenia występujące w trakcie pracy węzła obliczeniowego  
 Źródło: praca własna



Rys. 4.8. Maksymalne wartości funkcji przystosowania w każdej iteracji dla sieci konwolucyjnych - podejście nr  
 2  
 Źródło: praca własna



Rys. 4.9. Średnie wartości funkcji przystosowania w każdej iteracji dla sieci konwolucyjnych - podejście nr 2  
Źródło: praca własna

Algorytm rozwiążanie ma postać:  $\mathbf{x}^* = \begin{bmatrix} 12 \\ 21 \\ 23 \\ 31 \end{bmatrix}$  i po jednej epoce uczenia osiąga skuteczność na poziomie 47.51%. Niestety, nie jesteśmy w stanie stwierdzić, czy jest to minimum globalne bez wykonywania kosztownego przeszukiwania metodą siłową.

Do drugiego podejścia wymieniono całą infrastrukturę węzłów obliczeniowych, tj. zniszczono 20 maszyn wirtualnych opartych na obrazie *Data Science Virtual Machine for Linux* i wymieniono je na obrazy *Ubuntu Server 17.10*, jak opisano w 3.4.6. Stwierdzono, że na żadnej z maszyn nie występują ostrzeżenia jak na rys. 4.7. Dodatkowo, w stosunku do poprzedniej próby wprowadzono modyfikacje w celu zaoszczędzenia czasu obliczeniowego. Od tego momentu opisany w 3.4.3 rozpoczął wewnętrznie sprawdzać, czy powtarzają się osobniki do policzenia, aby nie liczyć ich kilka razy.

Dla drugiego uruchomienia przeprowadzono statystykę ile obliczeń zaoszczędzono. Sumarycznie nauczono 3827 różnych sieci. System zliczał liczbę duplikatów w każdej iteracji i przez całe swoje działanie zaoszczędził 501 obliczeń. W stosunku do pełnej liczby 4328 wyznaczeń funkcji przystosowania uzyskaliśmy 10% oszczędność czasu. Dodatkowo, algorytm tym razem nie liczył pełnych 100 iteracji, tylko zakończył swoje działanie w iteracji nr 68 (przy braku poprawy o zadany próg od iteracji nr 18), co ostatecznie zaowocowało ok. 15 h pracy przy drugiej próbie. Patrząc na wykres 4.8 widać, że poprawa wyniku pomiędzy 20 i 30 iteracją nie została uwzględniona jako poprawa, gdyż nie przekroczyła zadanego progu  $\epsilon$ .

#### **Porównanie wyników dla uczenia 1- i 10- epokowego**

W tym przypadku również przeprowadzono eksperyment dla dwóch przypadków - z infrastrukturą opartą na obrazach *Data Science Virtual Machine for Linux* oraz na *Ubuntu Server 17.10*. Plan eksperymentu przedstawia się następująco:

1. Z danych zebranych w badaniu 4.4 wybierz najlepszego, średniego i najsłabszego osobnika w każdej iteracji
2. Dokonaj uczenia owych osobników przez 10 epok

### 3. Dla każdej trójki najlepszy-średni-najgorszy sprawdź, czy kolejność została zachowana

Biorąc pod uwagę, że wszystkich możliwych ustawień trzech osobników jest  $P_3 = 3! = 6$ , jeżeli nie byłoby żadnego związku pomiędzy wynikiem uzyskanym po 1- a po 10- epokowym uczeniu, stosunek przypadków, w których kolejność została zachowana, do wszystkich przypadków powinien wynieść ok 16%.

Dla przypadku, tj. węzły obliczeniowe utworzone na podstawie *Data Science Virtual Machine for Linux* - jedynie w 3 przypadkach na 100 kolejność się zgadzała. Wynika to ze złego działania węzłów obliczeniowych - nie jesteśmy w stanie nic powiedzieć, ponieważ zbyt wiele wyników jest zniekształconych w wyniku braku pamięci. Dla uczenia 10-epokowego jedynie 9 sieci na 156 uczonych uzyskało wynik powyżej 15% - nie są to osiągi jakie powinna osiągać sieć po 10 epokach uczenia.

Dla przypadku drugiego, tj. infrastruktura oparta o *Ubuntu Server 17.10*, czyli, jak stwierdzono w 4.4 działającej metody obliczania wartości funkcji przystosowania, liczba iteracji dla których kolejność została zachowana wyniosła 40 na 68, co stanowi około 59%. Wynik ten jest dużo wyższy niż losowe ustalanie kolejności (poziom 16%). Pozwala to wysnuć wniosek, że istnieje pewien związek pomiędzy osiągami sieci po 1-epochie uczenia a ostatecznymi osiągami (tutaj rozumianymi jako 10-epokowe uczenie), gdyż szansa, że trzy sieci zachowają swoją kolejność pod względem jakości klasyfikacji wynosi niemalże 60 %. Co więcej, dla prostszego testu, to jest bez uwzględnienia środkowego środkowego osobnika, uszeregowanie zgadzało się w 100 % przypadków, tj. najlepsza sieć była najlepsza zarówno po 10 jak i 1 epochie uczenia, a najgorsza pozostała najgorszą.

Tutaj również, jak w 4.4 dokonano ulepszenia pomiędzy uruchomieniami algorytmu. W celu lepszej wizualizacji zachowania sieci w trakcie uczenia, zapisywano historię uczenia, tj. wartości funkcji przystosowania i funkcji kosztu w każdej epoce uczenia uzyskane na zbiorze uczącym oraz uzyskane na zbiorze testowym. Dzięki temu ulepszaniu można zobaczyć przebieg uczenia dowolnej sieci z arbitralnie wybranej iteracji algorytmu.

### **Porównanie przebiegu uczenia sieci w pierwszej i ostatniej iteracji algorytmu genetycznego**

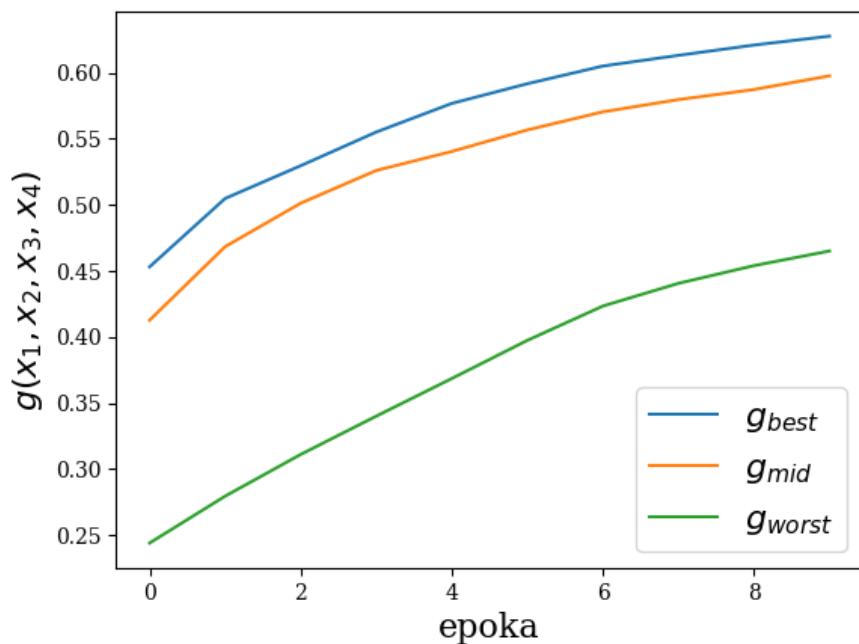
Celem tego badania jest sprawdzenie jak zmienia się dynamika uczenia sieci w zależności od jej struktury uzyskanej w wyniku działania algorytmu genetycznego. Aby zobrazować różnice pomiędzy owymi sieciami zestawiono ze sobą przebiegi uczenia:

1. Najlepszej, środkowej i najgorszej sieci z populacji inicjalnej (rys. 4.10)
2. J.w. sieci z populacji końcowej (rys. 4.11)
3. Najlepszej sieci z pierwszej i ostatniej iteracji algorytmu genetycznego (rys. 4.12)

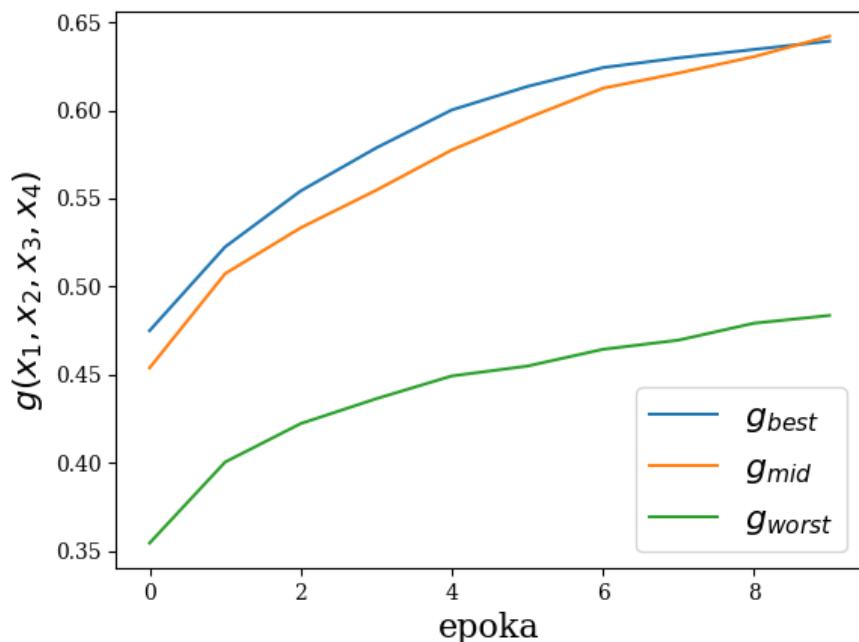
Co zaobserwowano:

- Wszystkie krzywe uczenia ewoluują w ten sam sposób - z każdą epoką uczenia coraz lepiej klasyfikują obrazy
- Porównując rys. 4.10 i rys. 4.11 widzimy, że w obydwu przypadkach najlepsza i średnia sieć znajdują się blisko siebie, a najgorsza znajduje się stosunkowo daleko pod nimi.
- Na rys. 4.11 widzimy sytuację, w której w dalszych fazach uczenia sieć średnia wyprzedza początkowo najlepszą. Zgodnie z 4.5, do takich sytuacji dochodzi w 40% iteracji algorytmu. Należy przy tym zauważyć, że przecinają się linia średnia z najlepszą, a linia najgorsza pozostaje w bezpiecznej odległości przez kolejne epoki. Nasuwa się przy tym wniosek, że im większa początkowa (po jednej epoce) różnica w jakości klasyfikacji obrazów przez sieć, tym większe prawdopodobieństwo, że po pełnym uczeniu relacja pomiędzy jakością sieci się nie zmieni.
- Porównując rys. 4.10 z rys. 4.11 lub patrząc na przebiegi na rys. 4.12 widzimy, że działanie algorytmu genetycznego podnosi pułap, na którym znajdują się krzywe uczenia. Wartości zarówno w 1. jak i 10. epoce znajdują się wyżej w iteracji ostatniej niż pierwszej.

Ostatecznie widoczne jest polepszenie procesu uczenia sieci przez algorytm genetyczny. Nie jest to polepszenie w sensie dynamicznym, gdyż kształt funkcji uczenia prawie się nie zmienia, jednakże w sensie statycznym, czyli wartości na początku i na końcu, widać poprawę.



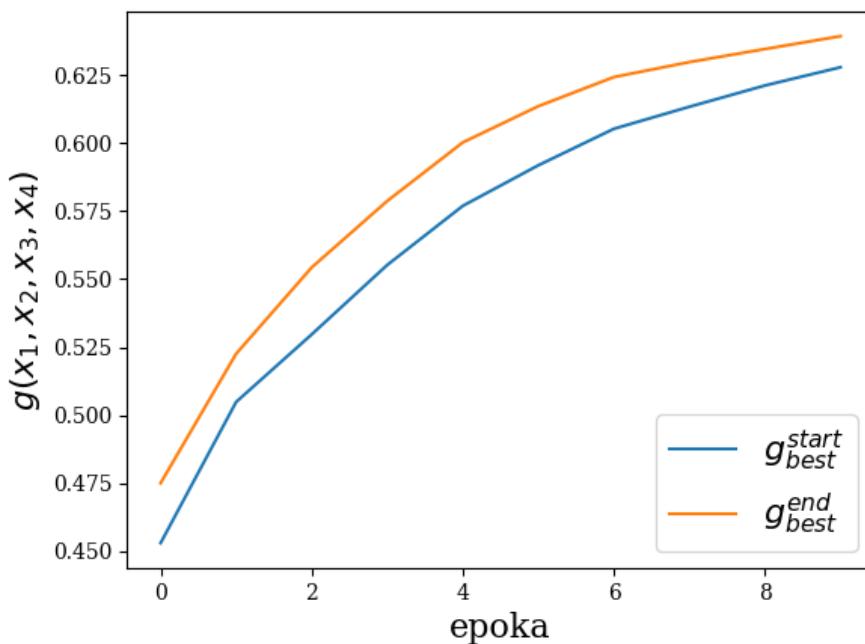
Rys. 4.10. Przebieg uczenia najlepszej, średniej i najgorszej sieci w pierwszej iteracji algorytmu genetycznego  
Źródło: praca własna



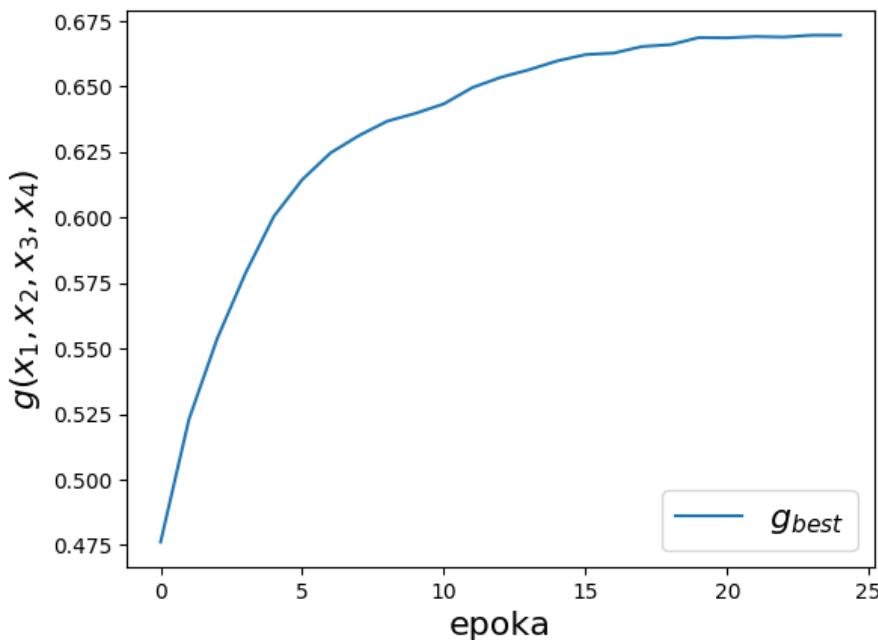
Rys. 4.11. Przebieg uczenia najlepszej, średniej i najgorszej sieci w ostatniej iteracji algorytmu genetycznego  
Źródło: praca własna

#### **Osiągi najlepszej według algorytmu sieci po 25 epokach uczenia**

Dla rozwiązania  $\mathbf{x}^* = \begin{bmatrix} 12 \\ 21 \\ 23 \\ 31 \end{bmatrix}$  przeprowadzone zostało uczenie 25-epokowe. Na rys. 4.13 przedstawiono krzywą uczenia dla tego przypadku:



Rys. 4.12. Przebieg uczenia najlepszej sieci w pierwszej i ostatniej iteracji algorytmu genetycznego  
 Źródło: praca własna



Rys. 4.13. Przebieg uczenia najlepszego osobnika z 4.4 dla 25 epok  
 Źródło: praca własna

Ostateczny wynik uzyskany przez sieć to 67%. Jest to gorszy wynik, niż w przykładzie, na którym oparto budowanie modeli sieci i uczenie ich (dostępny pod adresem [https://github.com/keras-team/keras/blob/master/examples/cifar10\\_cnn.py](https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py)) Powodem tego może być brak zastosowania warstw *dropout* w sieci modyfikowanej przez algorytm genetyczny, co prowadzi do nadmiernego dopasowania. Dalsza optymalizacja powyżej uzyskanych 67% wymagałaby modyfikacji elementów przyjętych jako stałe w strukturze sieci. Wynik ten jest jednak optymalny przy tak narzuconej strukturze.

## **Podsumowanie**

Badanie zależności pomiędzy początkową wydajnością uczonych sieci a ich wynikami po pełnym uczeniu okazało się kosztowne i czasochłonne.

Na samym początku wymagało pogłębiania wiedzy o:

- Uczeniu maszynowym
- Metauczeniu
- Uczeniu z nadzorem
- Klasyfikacji obrazków
- Zbiórce CIFAR-10
- Metodzie k-najbliższych sąsiadów
- Klasyfikatorach liniowych
- Sieciach neuronowych - ogólnie
- Konwolucyjnych sieciach neuronowych
- Algorytmie genetycznym

Z takim przygotowaniem teoretycznym można było podejść do zaprojektowania systemu do przeprowadzenia badań. To z kolei wiązało się z kolejnymi krokami:

1. Redukcji problemu, to jest dokładniejszego zdefiniowania problemu z uwzględnieniem ograniczonego czasu i budżetu na obliczenia
2. Zdefiniowania osobnika dla algorytmu genetycznego jako sieci neuronowej o określonej strukturze z pewnymi zmiennymi parametrami
3. Zdefiniowaniu konkrentycznych operacji genetycznych jakie przeprowadzane będą w tej konkretnej implementacji algorytmu genetycznego, a konkretnie zdefinowanie:
  - krzyżowania
  - mutacji
  - upewnienia się, że osobniki znajdują się wewnątrz zdefiniowanej przestrzeni poszukiwań
  - substytucji między pokoleniami
4. Zaprojektowaniu ogólnej architektury oprogramowania do przeprowadzenia eksperymentu
5. Zdefiniowaniu sposobu pracy węzła zarządzającego przeprowadzaniem eksperymentu
6. Zaprojektowaniu systemu kolejkowania zadań w celu przyspieszenia obliczeń
7. Stworzeniu protokołu do komunikacji z węzłami zdalnymi
8. Implementacji węzła obliczeniowego wykonującego zlecone mu zadania i zwracającego wyniki
9. Automatyzacji tworzenia nowych węzłów obliczeniowych
10. Analizie replikatywności wyników badań ze względu na korzystanie z generatora liczb pseudolosowych

Powstawanie owego systemu wymagało czasu, dogłębnego przemeślenia oraz pochłonęło zasoby na zarówno udane jak i niedane próby przeprowadzenia eksperymentu. Ostatecznie testy i wyniki, które zostały przeprowadzone można podsumować następująco:

- Przeprowadzono badanie wariancji wyników dla sieci o jendnej strukturze przy różnych wartościach ziarna generatora losowego w celu wyznaczenia progu poprawy pomiędzy iteracjami

- Przeprowadzono test działania algorytmu genetycznego dla testowej funkcji Levego
- Przeprowadzono test działania systemu kolejkowego dla jednej maszyny lokalnej
- Przeprowadzono poszukiwanie optymalnej struktury sieci neuronowej za pomocą algorytmu genetycznego w dwóch próbach
- Porównano wyniki dla uczenia 1- i 10- epokowego
- Zaobserwowano proces uczenia dla wybranych z pierwszej i ostatniej iteracji algorytmu genetycznego i porównano je
- Uzyskaną w wyniku działania algorytmu genetycznego sieć uczono przez 25 epok jako pełne uczenie

W toku przeprowadzonych badań udało się potwierdzić obydwie stawiane we wstępnie tezy, tj.

1. Możliwa jest optymalizacja struktury konwolucyjnych sieci neuronowych za pomocą algorytmu genetycznego
2. Istnieje związek pomiędzy skutecznością sieci po krótkim i długim uczeniu

System można w dalszym ciągu dopracować. Poniżej przedstawiono kilka propozycji ulepszeń:

- Aby uzyskać replikatywność badań zrezygnowano z równoległych obliczeń na GPU i system dostosowany został do obliczeń na CPU, możnaby jednak uwzględnić taką możliwość w kolejnych generacjach systemu.
- System kolejkowy możnaby dodatkowo usprawnić implementując algorytm rozdzielania zadań dla równoległych maszyn, np. implementując algorytm CDS (Campbella, Dudeka i Smitha)
- Można w dalszym ciągu ulepszyć czytelność kodu przepisując odpowiednie fragmenty
- Algorytmy parametry genetycznego dobrane zostały metodą prób i błędów - możnaby wejść na kolejny poziom metauczenia również dobierając jego parametry za pomocą innego algorytmu uczenia maszynowego
- Ustalone elementy sieci neuronowej (stałą część jej struktury) możnaby ulepszyć tak, by ostateczna wynikowa sieć miała lepsze wyniki klasyfikacji
- Węzły zdalne w żaden sposób nie sprawdzają, kto wysyła im zadania. Może to doprowadzić do sytuacji gdzie uruchomienie jednocześnie kilku systemów kolejkowych zlecających zadanie uniemożliwi poprawne działanie systemu. Należałoby to poprawić w przypadku korzystania z systemu przez więcej niż jedną osobę.

Dodatkowo pojawiło się nowe pytanie: jaki wpływ ma początkowa różnica w skuteczności klasyfikacji przez sieć jednoepokową na tę samą różnicę dla 10-epok? Widać zatem, że można dalej prowadzić badania w tym kierunku i pozostało wiele pola do badań w dziedzinie dobierania optymalnej struktury konwolucyjnych sieci neuronowych.

## Bibliografia

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [2] Lisa Arthur. What is big data? Dostęp: 26/08/2018 pod adresem <https://www.forbes.com/sites/lisaarthur/2013/08/15/what-is-big-data/>, 2013.
- [3] David Ascher, Paul F. Dubois, Konrad Hinsen, James Hugunin, and Travis Oliphant. *Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA, ucrl-ma-128569 edition, 1999.
- [4] Jon Beck, Cynthia Nottingham, Dan Lepow, Jean-Paul Connock, and Ralph Squillace. Gpu optimized virtual machine sizes. Dostęp: 29/06/2018 pod adresem <https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-gpu>, 2018.
- [5] Tomasz Biały. Algorytmy genetyczne. Materiały Wykładowe Politechniki Gdańskiej, 2012.
- [6] Pavel Brazdil, Christophe Giraud-Carrier, Carlos Soares, and Ricardo Vilalta. *Metalearning - Applications to Data Mining*. 01 2009.
- [7] François Chollet et al. Keras. <https://keras.io>, 2015.
- [8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.
- [9] Mikal Khoso. How much data is produced every day? Dostęp: 26/08/2018 pod adresem <https://www.northeastern.edu/levelblog/2016/05/13/how-much-data-produced-every-day/>, 2016.
- [10] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.
- [11] Frederic Lardinois. A look inside facebook's data center. Dostęp: 26/08/2018 pod adresem <https://techcrunch.com/gallery/a-look-inside-facebooks-data-center/>, 2016.
- [12] Christiane Lemke, Marcin Budka, and Bogdan Gabrys. Metalearning: a survey of trends and technologies. *Artificial Intelligence Review*, 44(1):117–130, Jun 2015.
- [13] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Cs231n: Convolutional neural networks for visual recognition 2016.
- [14] Tom Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.
- [15] Mehryar Mohri. *Foundations of machine learning*. MIT Press, Cambridge, MA, 2012.
- [16] Stuart Russell. *Artificial intelligence : a modern approach*. Prentice Hall, Upper Saddle River, NJ, 2010.
- [17] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.
- [18] S. Surjanovic and D. Bingham. Virtual library of simulation experiments: Test functions and datasets. Dostęp: 29/06/2018 pod adresem <http://www.sfu.ca/~ssurjano/levy.html>.
- [19] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.
- [20] Jacek Żurada, Mariusz Barski, and Wojciech Jędruch. *Sztuczne sieci neuronowe : podstawy teorii i zastosowania*. Warszawa : Wydawnictwo Naukowe PWN, 1996., 1996.

## Spis rysunków

1.1	Fragment centrum obliczeniowego Źródło: [11] . . . . .	8
1.2	Rekomendowana przez algorytmy uczenia maszynowego muzyka w serwisie Spotify Źródło: praca własna . . . . .	9
2.1	Komputerowa reprezentacja zdjęcia kota Źródło: praca własna na podstawie [13] . . . . .	11
2.2	Problemy w klasyfikacji obrazów Źródło: praca własna na podstawie [13] . . . . .	12
2.3	Przykładowe obrazki ze zbioru CIFAR-10 Źródło: <a href="https://www.cs.toronto.edu/~kriz/cifar.html">https://www.cs.toronto.edu/~kriz/cifar.html</a>	13
2.4	Ilustracja działania algorytmu k najbliższych sąsiadów w dwóch wymiarach Źródło: praca własna . . . . .	14
2.5	Schematyczna wizualizacja działania klasyfikatorów liniowych Źródło: praca własna na podstawie [13] . . . . .	15
2.6	Mapowanie 4 monochromatycznych pikseli do 3 klas Źródło: praca własna na podstawie [13] . . . . .	16
2.7	Pojedynczy 3-wejściowy neuron Źródło: praca własna na podstawie [13, 20] . . . . .	17
2.8	Schematyczne przedstawienie architektury konwolucyjnej sieci neuronowej Źródło: [13]	19
2.9	Podział metod optymalizacji Źródło: [5] . . . . .	20
2.10	Schemat działania algorytmu genetycznego Źródło: praca własna . . . . .	22
3.1	Przedstawienie pojedynczego osobnika Źródło: praca własna . . . . .	24
3.2	Krzyżowanie jednorodne Źródło: praca własna . . . . .	25
3.3	Całokształt systemu przeprowadzania eksperymentu Źródło: praca własna . . . . .	26
3.4	Struktura katalogów projektu systemu eksperymentu Źródło: praca własna . . . . .	26
3.5	System kolejkowania obliczeń Źródło: praca własna . . . . .	29
3.6	Schemat wymiany komunikatów pomiędzy systemem kolejkowym a węzłami obliczeniowymi Źródło: praca własna . . . . .	31
3.7	Schemat systemu obsługi zadań Źródło: praca własna . . . . .	33
3.8	Interfejs Microsoft Azure - tworzenie maszyny wirtualnej Źródło: praca własna . . . . .	35
3.9	6 węzłów obliczeniowych w trakcie pracy Źródło: praca własna . . . . .	36
4.1	Wykres funkcji Levego Źródło: [18] . . . . .	39
4.2	Maksymalne wartości funkcji przystosowania w każdej iteracji dla funkcji Levego Źródło: praca własna . . . . .	40
4.3	Średnie wartości funkcji przystosowania w każdej iteracji dla funkcji Levego Źródło: praca własna . . . . .	40
4.4	Test działania całego systemu w ramach maszyny lokalnej Źródło: praca własna . . . . .	41
4.5	Maksymalne wartości funkcji przystosowania w każdej iteracji dla sieci konwolucyjnych - podejście nr 1 Źródło: praca własna . . . . .	42
4.6	Średnie wartości funkcji przystosowania w każdej iteracji dla sieci konwolucyjnych - podejście nr 1 Źródło: praca własna . . . . .	43
4.7	Ostrzeżenia występujące w trakcie pracy węzła obliczeniowego Źródło: praca własna . . . . .	44
4.8	Maksymalne wartości funkcji przystosowania w każdej iteracji dla sieci konwolucyjnych - podejście nr 2 Źródło: praca własna . . . . .	44
4.9	Średnie wartości funkcji przystosowania w każdej iteracji dla sieci konwolucyjnych - podejście nr 2 Źródło: praca własna . . . . .	45
4.10	Przebieg uczenia najlepszej, średniej i najgorszej sieci w pierwszej iteracji algorytmu genetycznego Źródło: praca własna . . . . .	47
4.11	Przebieg uczenia najlepszej, średniej i najgorszej sieci w ostatniej iteracji algorytmu genetycznego Źródło: praca własna . . . . .	47
4.12	Przebieg uczenia najlepszej sieci w pierwszej i ostatniej iteracji algorytmu genetycznego Źródło: praca własna . . . . .	48
4.13	Przebieg uczenia najlepszego osobnika z 4.4 dla 25 epok Źródło: praca własna . . . . .	48

## **Spis tabel**

Maszyny wirtualne zoptimalizowane pod kątem głębokiego uczenia. Źródło: [4] . . . . 34

## **Skrótowy opis dyplomu**

### **Tytuł dyplomu**

Optymalizacja struktury konwolucyjnych sieci neuronowych za pomocą algorytmu genetycznego

### **Cel i przeznaczenie aplikacji**

Stworzenie systemu umożliwiającego badanie możliwości użycia algorytmu genetycznego do znalezienia optymalnej struktury dla konwolucyjnej sieci neuronowej w sensie skuteczności klasyfikacji obrazów.

### **Funkcjonalność**

#### *Spis realizowanych funkcji*

Wytwarzony system realizuje następujące funkcje:

- Generacja sieci konwolucyjnych o różnych w sensie ilości neuronów w poszczególnych warstwach strukturach
- Rodzielanie zadań uczenia sieci przez system kolejkowy do poszczególnych węzłów obliczeniowych
- Uczenie sieci o zadanej strukturze, liczbie epok uczenia oraz przy ustalonym ziarnie generatora liczb losowych
- Przetwarzanie uzyskanych w wyniku przeprowadzonego eksperymentu danych do czytelnej formy wykresów

#### *Lista przykładowych zastosowań*

Przykładowe zastosowania wytworzonego systemu:

- Badanie czy dany algorytm (w tym konkretnym przypadku algorytm genetyczny) może posłużyć do optymalizacji struktury sieci neuronowej
- Badanie zależności pomiędzy krótkim (np. przez 1 epokę) uczeniem sieci neuronowych a uczeniem długim (np. przez 10 epok)

#### *Szczegółowy opis działania aplikacji*

Algorytm genetyczny generuje potencjalnie optymalne struktury sieci neuronowych. Wygenerowane sieci oceniane są pod względem skuteczności klasyfikacji obrazków na zbiorze testowym CIFAR-10. Wynikowo uzyskuje się optymalną pod względem struktury sieć neuronową oraz dużo danych z przebiegu algorytmu genetycznego. Dane są następnie ręczne przetwarzane z wykorzystaniem pomocniczych skryptów w języku *Python*.

### **Ogólna architektura systemu**

System składa się z węzła zarządzającego przetwarzanymi eksperymentami i węzłów obliczeniowych, wykonujących powierzone im zadania. Każdy z nich z założenia jest komputerem z zainstalowanym interpreterem języka *Python* w wersji 3 wraz z odpowiednimi bibliotekami: *Keras*, *TensorFlow*, *Numpy*. System komunikuje się przez sieć przy pomocy interfejsu *socket* poprzez protokół TCP/IP.

### **Architektura sprzętu**

Projekt został w głównej mierze oparty o technologie wirtualizacji, zatem architektura sprzętu nie jest w dużej mierze istotna. Podane zostaną sprzętowe parametry użytych jako węzły obliczeniowe maszyn wirtualnych: 1 vCPU, 3.5 GB pamięci RAM, 7 GB pamięci SSD.

## **Architektura oprogramowania**

Oprogramowanie zostało podzielone na logicznie separowalne moduły. Jeden z nich jest odpowiedzialny za działanie węzłów zarządzających i zawiera w sobie wszystkie pożyteczne dla takiego węzła funkcjonalności, np. system kolejkowy. Drugi odpowiedzialny jest za pracę węzła obliczeniowego, który jest w istocie komputerem z jednocześnie pracującymi dwoma programami - jednym nasłuchującym zadań, drugim nasłuchującym zapytań o wyniki. Trzeci moduł zawiera oprogramowanie pomocnicze, służące automatyzacji procesu tworzenia nowych węzłów obliczeniowych.

## **Opis sposobu wytwarzania aplikacji**

### **Założenia i sformułowanie zadania**

System powinien umożliwić zbadanie przystosowania algorytmu genetycznego do poszukiwania optymalnej struktury konwolucyjnej sieci neuronowej oraz zbadanie związku pomiędzy uczeniem krótkim i długim.

### **Analiza problemu (z przeglądem dotychczasowych rozwiązań)**

Dokonano analizy problemu z uwzględnieniem następujących zagadnień: uczenie maszynowe, uczenie z nadzorem, metauczenie, klasyfikacja obrazów, klasyfikatory liniowe, konwolucyjne sieci neuronowe, algorytm genetyczny.

### **Specyfikacje**

Specyfikacje w głównej mierze narzucone były przez ograniczony czas i budżet, zatem dążono do stworzenia systemu który wykona obliczenia małym kosztem i możliwie szybko.

### **Przygotowanie projektu**

Projekt przygotowano odpowiednio przygotowując w odpowiedniej kolejności projekt systemu kolejkowego, węzła obliczeniowego oraz węzła zarządzającego.

### **Prototypowanie i implementacja**

Prototypowanie i implementacja odbyły się w podobnej kolejności, w jakiej przygotowano projekt - stworzono prototyp systemu kolejkowego, węzła obliczeniowego i zarządzającego, a implementacja postępowała wraz z kolejnymi ulepszeniami dokonywanymi na prototypach.

### **Testowanie**

Testowano zarówno każdy moduł oddzielnie jak i integrację wszystkich modułów.

### **Ocena aplikacji (w tym porównanie do innych rozwiązań)**

Aplikacja spełnia postawione przed nią zadanie. W porównaniu do istniejących rozwiązań jest rozszerzona o możliwość badania związku pomiędzy uczeniem krótkim i długim oraz korzysta ze zrównoleglenia obliczeń przy pomocy systemu kolejkowego.

### **Wnioski i perspektywy dalszych prac**

Wnioski są następujące -

- można poszukiwać struktury konwolucyjnych sieci neuronowych przy pomocy algorytmu genetycznego
- istnieje związek pomiędzy uczeniem krótkim i długim sieci neuronowych

Dalsze prace polegałyby na dopracowywaniu systemu do szybszego i stabilniejszego działania, projektowaniu nowych eksperymentów oraz uruchamianie ich na wytworzonym systemie oraz dokładniejszego zbadania zależności pomiędzy różnicą w wynikach po uczeniu krótkim a po uczeniu długim.

## Kod programu

Poniżej zamieszczono kod źródłowy najważniejszych części systemu:  
Węzeł zarządzający:

code/genetic\_algorithm.py

```
import random
import copy
import master.queue_handling as qh
from master.csv_handling import *
import common.params as p
from common.helper_classes import Job, Individual

def population_init(N, m, layers):
    population = []
    for i in range(N):
        population.append(Individual.random(m, layers))
    return population

def select_one(population):
    total = sum(map(lambda x: x.score, population))
    pick = random.uniform(0, total)
    current = 0
    for individual in population:
        current += individual.score
        if current > pick:
            return individual

def selection(population):
    """
    Implements proportional selection
    :param population: population with scores evaluated
    :return: parents pool
    """
    parents = []
    for i in range(len(population)):
        parents.append(copy.deepcopy(select_one(population)))
    return parents

def pair_crossover(p1, p2, m=p.m):
    assert len(p1.phenotype) == len(p2.phenotype)

    n = len(p1.phenotype)

    mask = random.randint(1, 2 ** (m * n) - 1)
    invmask = 2 ** (m * n) - 1 - mask

    p1code = phenotype_to_genotype(p1.phenotype, n, m)
    p2code = phenotype_to_genotype(p2.phenotype, n, m)

    d1code = p1code & invmask | p2code & mask
    d2code = p2code & invmask | p1code & mask
```

```

d1 = genotype_to_phenotype(d1code, n, m)
d2 = genotype_to_phenotype(d2code, n, m)

    return Individual(d1), Individual(d2)

def random_index_pairs(n):
    indexes = list(range(n))
    random.shuffle(indexes)
    return [[indexes[i], indexes[i+1]] for i in range(0, n, 2)]

def crossover(population, crossover_prob):
    new_population = []
    for i, j in random_index_pairs(len(population)):
        if random.random() < crossover_prob:
            new_population.extend(pair_crossover(population[i],
                                                   population[j]))
        else:
            new_population.append((population[i], population[j]))
    return new_population

def perform_mutation(individual, m=p.m):
    assert(isinstance(individual, Individual))
    layers = len(individual.phenotype)
    which_layer = random.randint(0, layers - 1)
    new_genotype = list(individual.phenotype)
    new_genotype[which_layer] = random.randint(1, 2*m-1)
    individual.phenotype = tuple(new_genotype)

def mutation(population, mutation_prob=p.mutation_prob):
    for individual in population:
        if random.random() < mutation_prob:
            perform_mutation(individual)

def fix_broken_individual(individual):
    new_phenotype = []
    for layer in individual.phenotype:
        if layer < 1:
            new_phenotype.append(1)
        else:
            new_phenotype.append(layer)
    individual.phenotype = tuple(new_phenotype)

def fix_broken_population(population):
    for individual in population:
        if 0 in individual.phenotype:
            fix_broken_individual(individual)

def evaluate_scores(population, workmanager, computed_scores,
                   m=5, layers=4, seed=1337):
    jobs = []
    computed = []

```

```

for individual in population:
    get_already_computed(individual, computed_scores, m, layers)
    if not individual.score:
        jobs.append(Job(individual, 1, seed))
    else:
        computed.append(individual)
computed.extend(workmanager.evaluate(jobs))
return computed

def restore_state_from_file(read_csv, m, layers, N):
    fresh_start = (False, 1, 0, 0, None)

    if not os.path.isfile(read_csv):
        return fresh_start

    with open(read_csv, 'r') as f:
        reader = csv.reader(f)

        iters, codes, scores, losses = [], [], [], []

        next(reader, None) # skip the header
        for row in reader:
            it, code, score, loss = row
            iters.append(int(it))
            codes.append(int(code))
            scores.append(float(score))
            losses.append(float(loss))

    if not iters: # nothing to read
        return fresh_start

    it = max(iters)
    best_score = max(scores)

    for i, score in enumerate(scores):
        if score == best_score:
            first_max_it = iters[i]
            break
    no_improvement = it - first_max_it

    population = []
    for code in codes[-N:]:
        population.append(Individual(genotype_to_phenotype(code,
                                                          layers, m)))

    return True, it, no_improvement, best_score, population

def genetic_operations(population, crossover_prob, mutation_prob, keep):
    parents = selection(population)
    descendants = crossover(parents, crossover_prob)
    mutation(descendants, mutation_prob)
    fix_broken_population(descendants)
    random.shuffle(descendants)
    population[keep:] = descendants[keep:]
    return population

```

```

def run():
    random.seed(p.seed)
    create_csv_if_not_existent(p.genetic_write_csv)
    computed_scores = \
        load_already_computed_from_file(p.genetic_read_csv)
    wm = qh.WorkManager()
    restored, it, no_improvement, previous_best_score, population = \
        restore_state_from_file(p.genetic_read_csv, p.m, p.layers, p.N)
    if not restored:
        population = population_init(p.N, p.m, p.layers)
    elif it == p.max_iter:
        print("Restored fully conducted experiment. Quitting.")
        return

    while it <= p.max_iter and no_improvement < p.max_no_improvement:
        print("----Iter: {} / {} ----".format(it, p.max_iter))
        population = evaluate_scores(population, wm, computed_scores,
                                      p.m, p.layers, p.seed)
        population = sorted(population,
                            key=lambda individual: individual.score,
                            reverse=True)
        save_csv_and_history(p.genetic_write_csv,
                             population, it, p.layers, p.m)
        best_score = max(map(lambda x: x.score, population))
        improvement = best_score - previous_best_score
        if improvement > p.eps:
            no_improvement = 0
            previous_best_score = best_score
        else:
            no_improvement += 1
        population = genetic_operations(population, p.crossover_prob,
                                         p.mutation_prob, p.keep)
        it += 1

if __name__ == '__main__':
    run()

```

code/queue\_handling.py

```

import sys
import time
import abc
import socket
import pickle
import copy
from math import sin, pi
from common.helper_classes import Job, Individual

sys.path.append("..")

job_port = 4123
result_port = 4124

def host_available(host):
    try:
        for port in job_port, result_port:

```

```

        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(1)
        sock.connect((host, port))
        sock.send(b"UP?")
        response = sock.recv(1024)
        sock.close()
        if not b"UP" in response:
            return False
    except Exception as e:
        # print('Worker %s not available.' % host)
        return False
    return True

class TimeoutException(Exception):
    pass

class Worker(object):
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def is_available(self):
        return

    @abc.abstractmethod
    def assign_job(self, job):
        """
        Returns True if job successfully assigned, False otherwise
        :rtype: bool
        """
        return

    @abc.abstractmethod
    def has_result(self):
        return

    @abc.abstractmethod
    def get_result(self):
        return

    @abc.abstractmethod
    def free_worker(self):
        return

    @abc.abstractmethod
    def get_current_job(self):
        return

class RemoteWorker(Worker):
    def __init__(self, ip):
        self.available = True
        self.result = None
        self.job = None

        self.ip = ip
        self._job_port = job_port

```

```

        self._result_port = result_port
        self._start = None
        self.timeout = 4800
        # after 1,5 hour something is wrong in 10 epoch learning

    def __eq__(self, other):
        return self.ip == other.ip

    def __str__(self):
        return "Available:{}|Result:{}|Job:{}|IP:{}"\ \
            .format(self.available, self.result, self.job, self.ip)

    def is_available(self):
        return self.available

    def assign_job(self, job):
        self.available = False
        self.job = job
        self._start = time.time()
        try:
            self._remote_dispatch(job)
        except Exception as e:
            print(e)
            print("Job_dispatch_failed.")
            return False
        return True

    def has_result(self):
        if time.time() - self._start > self.timeout:
            raise TimeoutException
        result = self._fetch_remote_result()
        if result:
            self.result = result
            return True
        return False

    def get_result(self):
        return self.result

    def free_worker(self):
        self.available = True
        self.result = None

    def get_current_job(self):
        assert self.job is not None
        return self.job

    def _remote_dispatch(self, job):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((self.ip, self._job_port))
        s.sendall(pickle.dumps(job))
        s.close()

    def _fetch_remote_result(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((self.ip, self._result_port))
        s.sendall(b"RDY?")
        response = s.recv(8192)

```

```

    if "BSY" in str(response):
        return None
    else:
        result = pickle.loads(response)
        assert(result.score is not None)
        return result

class LevyWorker(Worker):
    def __init__(self):
        self.available = True
        self.result = None
        self.job = None

    def is_available(self):
        return self.available

    def assign_job(self, job):
        self.job = job
        self.available = False
        xm = job.individual.phenotype
        x = (xm[0] - 3, xm[1] - 7, xm[2] - 13, xm[3] - 20)
        w = []
        for i in range(len(x)):
            w.append(1 + (x[i] - 1) / 4)
        middle_term = 0
        for i in range(len(x) - 1):
            middle_term += (w[i] - 1)**2 * (1 + 10 * sin(pi*w[i] + 1)
                                              * sin(pi*w[i] + 1))
        result = sin(pi*w[0]) * sin(pi*w[0]) + middle_term +\
                 (w[-1] - 1)**2 * (1 + sin(2*pi*w[-1]) * sin(2*pi*w[-1]))
        job.individual.loss = result
        job.individual.score = 1000 - result
        job.individual.training_history = "DUMMY_HISTORY"
        self.result = job.individual
        return True

    def has_result(self):
        if self.result:
            return True
        else:
            return False

    def get_result(self):
        return self.result

    def free_worker(self):
        self.available = True
        self.result = None
        self.job = None

    def get_current_job(self):
        return self.job

class WorkManager(object):
    def __init__(self):
        self.workers = []
        self.results = []

```

```

def __add_workers(self):
    hosts_pool = []
    with open("workers", "r") as f:
        for line in f:
            hosts_pool.append(line.strip())
    for host in hosts_pool:
        if not RemoteWorker(host) in self.workers:
            if host_available(host):
                print("Adding_new_worker: %s" % host)
                self.workers.append(RemoteWorker(host))

def evaluate(self, jobs):
    """
    Evaluates a given list of jobs, avoiding calculating duplicates.
    :param jobs: list of jobs with scores to be computed
    :return: list of individuals with scores computed
    """
    self.results = []

    job_keys = set(map(lambda x: x.get_key(), jobs))

    inner_jobs = []

    for phenotype, epochs, seed in job_keys:
        individual = Individual(phenotype)
        j = Job(individual, epochs, seed)
        inner_jobs.append(j)

    inner_results = {}

    expected_results = len(inner_jobs)
    while len(inner_results) < expected_results:
        # makes adding new workers while the script
        # is running possible
        self.__add_workers()
        for worker in self.workers[:]:
            if worker.is_available() and inner_jobs:
                job = inner_jobs.pop()
                print("Worker_available...Assigning_job_(%d/%d)...%s"
                    .format(expected_results - len(inner_jobs),
                           expected_results, job))
                success = worker.assign_job(job)
                if not success:
                    print("Worker_%s_broken_in_assigning_job,"
                            "removing_from_pool_and_reassigning_job"
                            "%s worker.ip")
                    inner_jobs.append(worker.get_current_job())
                    self.workers.remove(worker)
        for worker in self.workers[:]:
            try:
                if not worker.is_available():
                    if worker.has_result():
                        result = worker.get_result()
                        job = worker.get_current_job()
                        inner_results[job.get_key()] = result
                        worker.free_worker()
            except Exception as e:

```

```

        print(e)
        print("Worker %s broken in result collection ,"
              " removing from pool and reassigning job"
              % worker.ip)
        inner_jobs.append(worker.get_current_job())
        self.workers.remove(worker)
        time.sleep(1)
    for job in jobs:
        key = job.get_key()
        result_copy = copy.deepcopy(inner_results[key])
        self.results.append(result_copy)

    return self.results

```

code/csv\_handling.py

```

# This file contains all functions related to csv file handling
# Each function should specify what kind of csv file it expects
from common.helper_classes import Individual
import os
import csv


def get_set_of_codes_from_filename(csv_filename, N):
    """
    Gets all unique codes from a csv file containing first experiment
    results
    :param N: number of entries per population
    :param csv_filename: the csv file in the form:
                         iter, code, score, loss
    :return: set of unique codes for second experiment
    """
    results = []

    with open(csv_filename, "r") as f:
        for i, line in enumerate(csv.reader(f)):
            results.append(line)

    number_of_iterations = int(results[-1][0])
    # -1 for last entry, 0 for first field meaning iteration number

    best_indexes = [1 + N * i for i in range(number_of_iterations)]
    middle_indexes = [N//2 + N * i for i in range(number_of_iterations)]
    worst_indexes = [N*(i+1) for i in range(number_of_iterations)]

    indexes = best_indexes + middle_indexes + worst_indexes
    codes_to_calculate = []

    for i in indexes:
        codes_to_calculate.append(int(results[i][1]))

    return set(codes_to_calculate)

def phenotype_to_genotype(phenotype, n, m):
    """
    Returns the binary representation of the phenotype
    :param phenotype: tuple containing the layer filter numbers
    """

```

```

:param n: number of layers
:param m: number of bits to encode numbers of layers on
:return: genotype : binary representation of the phenotype
"""
code = 0
for i in range(n):
    code += phenotype[i]
    code = code << m
code = code >> m
return code

def genotype_to_phenotype(genotype, n, m):
    """
    Returns the tuple representation of the genotype
    :param genotype : binary representation of the phenotype
    :param n: number of layers
    :param m: number of bits to encode numbers of layers on
    :return: phenotype: tuple containing the layer filter numbers
    """
    phenotype = []
    for i in range(n):
        phenotype.append(genotype & (2 ** m - 1))
        genotype = genotype >> m
    return tuple(phenotype[::-1])

def map_results_to_dicts(results, iteration, n, m):
    """
    Prepares the current iteration's population for csv writing
    :param n:
    :param m:
    :param iteration: iteration number
    :param results: list of Individuals
    :return: list of dicts containing rows for csv writing
    """
    dicts = []
    for result in results:
        assert(isinstance(result, Individual))
        dicts.append({"iter": iteration,
                      "code": phenotype_to_genotype(result.phenotype,
                                                    n, m),
                      "score": result.score,
                      "loss": result.loss})
    return dicts

def load_already_computed_from_file(read_csv):
    """
    Restores the already stored results in case of resuming
    the experiment
    :param read_csv: csvfile to read already stored entries
    :return: a dictionary with already computed
    """
    if not os.path.isfile(read_csv):
        return {}
    reader = csv.reader(open(read_csv, 'r'))
    computed_scores = {}

```

```

for row in reader:
    iteration, code, score, loss = row
    computed_scores[code] = (score, loss)
return computed_scores

def get_already_computed(individual, computed_scores, m=5, layers=4):
    """
    Modifies an individual in place copying the score from provided
    dictionary with scores
    :param m:
    :param layers:
    :param individual: individual to retrieve the score for
    :param computed_scores: dict containing results, key is genotype,
        value is (score, loss)
    :return: none, individual modified in place
    """
    key = str(phenotype_to_genotype(individual.phenotype, layers, m))
    if key in computed_scores:
        scores = computed_scores[key]
        individual.score, individual.loss = map(float, scores)

def save_csv_and_history(write_csv, results, iteration, n, m):
    """
    Appends current genetic algorithm
    :param n:
    :param m:
    :param write_csv: filename to append the current iteration results
        to
    :param results: current population of individuals
    :param iteration: current iteration number
    :return:
    """
    dicts = map_results_to_dicts(results, iteration, n, m)
    with open(write_csv, 'a') as csv_file:
        fieldnames = ['iter', 'code', 'score', 'loss']
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
        for entry in dicts:
            writer.writerow(entry)
    with open(write_csv[:-4] + "_history.txt", "a") as f:
        for result in results:
            line = "{:18}|{}|{}\n".format(str(result.phenotype),
                                         result.training_history)
            f.write(line)

def create_csv_if_not_existent(write_csv):
    if not os.path.isfile(write_csv):
        with open(write_csv, 'w') as csvfile:
            fieldnames = ['iter', 'genotype', 'score', 'loss']
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
            writer.writeheader()

```

Węzeł obliczeniowy:

code/listener.jobs.py

```
import sys
```

```

# for importing helper classes
sys.path.append("..")

import socket
import pickle
from common.helper_classes import Job
import eval_cnn

HOST = ''          # Symbolic name meaning all available interfaces
PORT = 4123        # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
while 1:
    conn, addr = s.accept()
    data = conn.recv(1024)
    if b"UP?" in data:
        conn.send(b"UP")
        conn.close()
        continue
    conn.close()
    training_data = pickle.loads(data)
    assert(isinstance(training_data, Job))
    eval_cnn.eval_network(training_data.individual,
                          training_data.epochs,
                          training_data.seed)

```

code/listener\_results.py

```

import socket
import os

HOST = ''          # Symbolic name meaning all available interfaces
PORT = 4124        # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
while 1:
    conn, addr = s.accept()
    data = conn.recv(1024)
    if b"UP?" in data:
        conn.send(b"UP")
    elif b"RDY?" in data:
        if not os.path.isfile("result"):
            conn.send(b"BSY")
        else:
            with open("result", "rb") as f:
                res = f.read()
                conn.sendall(res)
            os.remove("result")
    conn.close()
s.close()

```

code/eval\_cnn.py

```

'''Trains a single convolutional neural network for specified epochs
number with a specified random seed
'''

from keras.layers import Dense, Activation, Flatten

```

```

from keras.layers import Conv2D, MaxPooling2D
from keras.models import Sequential
from keras.datasets import cifar10
from keras import backend as K
import tensorflow as tf
from common import helper_classes
import random as rn
import numpy as np
import pickle
import keras
import os

def set_random_seed(seed):
    os.environ['PYTHONHASHSEED'] = '0'
    np.random.seed(seed)
    rn.seed(seed)
    # turn off parallel computing for reproductivity
    session_conf = tf.ConfigProto(intra_op_parallelism_threads=1,
                                    inter_op_parallelism_threads=1)
    tf.set_random_seed(seed)
    sess = tf.Session(graph=tf.get_default_graph(),
                      config=session_conf)
    K.set_session(sess)

def eval_network(individual, epochs, seed):
    assert(isinstance(individual, helper_classes.Individual))
    set_random_seed(seed)
    batch_size = 32
    num_classes = 10

    print("Layers: " + str(individual.phenotype))
    # The data, split between train and test sets:
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()

    # Convert class vectors to binary class matrices.
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)

    layers = individual.phenotype
    model = Sequential()
    for idx, layer in enumerate(layers):
        if idx == 0:
            model.add(Conv2D(layer, (3, 3), padding='same',
                            input_shape=x_train.shape[1:]))
            model.add(Activation('relu'))
        else:
            model.add(Conv2D(layer, (3, 3)))
            model.add(Activation('relu'))
            # add 1 pooling layer in the middle
            if (idx + 1) == (len(layers) // 2):
                model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128))
    model.add(Dense(num_classes))
    model.add(Activation('softmax'))

```

```

# initiate RMSprop optimizer
opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)

# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

history = model.fit(x_train, y_train,
                     batch_size=batch_size,
                     epochs=epochs,
                     validation_data=(x_test, y_test),
                     shuffle=False)

# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=1)

print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
individual.loss = scores[0]
individual.score = scores[1]
individual.training_history = str(history.history)
with open("result", 'wb') as f:
    pickle.dump(individual, f)

```

code/starter.sh

```

rm result
pkill python
python3 listener_results.py > /dev/null &
disown
while :
do
    python3 listener_jobs.py > /dev/null
    echo "Job listener failed, restarting in 10 seconds"
    sleep 10
done

```

## Instrukcja obsługi systemu

System uruchamia się w sposób następujący:

1. Przygotowuje się węzły obliczeniowe w następujący sposób:
  1. 1. Na dowolnej platformie chmurowej tworzy się węzły obliczeniowe, czyli maszyny wirtualne. Zaleca się użycie podobnych maszyn do użytych w toku badań, tj. maszyny z 1 vCPU, 3.5 GB pamięci ram, 7 GB pamięci masowej, system operacyjny Ubuntu 17.10
  1. 2. Dla każdej maszyny wirtualnej otwiera się wymagane do działania systemu porty TCP 4123-4124 dla połączeń przychodzących
  1. 3. Instaluje się oprogramowanie wymagane przez system, tj. intrepreter języka Python w wersji 3, instalator bibliotek pip, biblioteki TensorFlow, Keras, Numpy
  1. 4. Pobiera się kod systemu z przygotowanego repozytorium
  1. 5. Uruchamia się skrypt starter.sh w tle, który uruchamia odpowiednie skrypty w języku Python i zapewnia restart systemu w razie awarii
2. Do pliku workers dopisuje się adresy ip węzłów obliczeniowych, po jednym na każdą linię pliku
3. Lokalnie uruchamia się węzeł zarządzający reprezentujący jeden eksperyment. W repozytorium pliki, które są do tego przeznaczone, to:
  - *genetic\_algorithmm.py*
  - *full\_learning.py*
  - *25epoch.py*
4. dokonuje się analizy zebranych w plikach csv danych przy pomocy pozostałych zgromadzonych w repozytorium skryptów

W wyniku takiego działania otrzymuje się wyniki i wykresy identyczne jak dla przeprowadzonych w rozdziale 4 badaniach, poza opisanymi błędymi próbami, które zostały poprawione w toku pracy nad systemem. W celu dojścia do błędnych prób zaleca się przejrzenie historii systemu kontroli wersji *git* dla repozytorium *gacnn* w serwisie *GitHub* pod adresem <https://github.com/opiechow/gacnn>.

## Instrukcja dla projektanta

Podobnie jak w przypadku zwykłego uruchomienia systemu, uruchamia się system w stan gotowości do pracy, tj.

1. Przygotowuje się węzły obliczeniowe w następujący sposób:
  1. 1. Na dowolnej platformie chmurowej tworzy się węzły obliczeniowe, czyli maszyny wirtualne. Zaleca się użycie podobnych maszyn do użytych w toku badań, tj. maszyny z 1 vCPU, 3.5 GB pamięci ram, 7 GB pamięci masowej, system operacyjny Ubuntu 17.10
  1. 2. Dla każdej maszyny wirtualnej otwiera się wymagane do działania systemu porty TCP 4123-4124 dla połączeń przychodzących
  1. 3. Instaluje się oprogramowanie wymagane przez system, tj. interpreter języka Python w wersji 3, instalator bibliotek pip, biblioteki TensorFlow, Keras, Numpy
  1. 4. Pobiera się kod systemu z przygotowanego repozytorium
  1. 5. Uruchamia się skrypt starter.sh w tle, który uruchamia odpowiednie skrypty w języku Python i zapewnia restart systemu w razie awarii
2. Do pliku workers dopisuje się adresy ip węzłów obliczeniowych, po jednym na każdą linię pliku

Następnie można przystąpić do projektowania nowych eksperymentów, czyli nowych implementacji węzła zarządzającego. Polega to na odpowieniu wykorzystaniu gotowego systemu kolejkowego, które przeprowadza się w sposób następujący:

1. W kodzie węzła zarządzającego należy utworzyć obiekt klasy *WorkManager*
2. Należy przygotować listę zadań, która jest listą obiektów klasy *Job* - dla każdego zadania należy ustalić osobnika klasy *Individual* do uczenia, liczbę epok uczenia oraz ziarno dla generatora liczb losowych
3. Korzystając z metody *evaluate(jobs)* obiektu klasy *WorkManager* należy przeprowadzić obliczenia
4. Uzyskaną listę wyników, czyli listę obiektów klasy *Individual* z uzupełnionymi wszystkimi polami należy zapisać do pliku, np. korzystając z funkcji *save\_csv\_and\_history(write\_csv, results, iteration, n, m)*.

W ten sposób można w szybki sposób dokonywać uczenia wielu sieci na równoległych maszynach. Przykładowe pliki, na których można wzorować nowe implementacje węzłów zarządzających:

- *genetic\_algorithmm.py*
- *full\_learning.py*
- *25epoch.py*