Karol Ziarek 318749
Jakub Kieruczenko 318669

POP 23Z

Zadanie 4.

Opracuj i przeprowadź eksperyment, w którym porównasz dowolny wariant ewolucji różnicowej z dowolnym wariantem strategii ewolucyjnej w ramach zadania uczenia sieci neuronowej. Ponadto porównaj działanie metod ewolucyjnych z wybraną metodą gradientową.

1. Opis analizowanego problemu

Celem projektu jest zbadanie i porównanie skuteczności ewolucji różnicowej i strategii ewolucyjnej z podejściem gradientowym w uczeniu sztucznych sieci neuronowych.

Uczenie sieci neuronowej opiera się na dostosowywaniu wag w kolejnych warstwach sieci neuronowej w celu minimalizacji funkcji straty, co odbywa się na podstawie danych treningowych. W klasycznym podejściu, gdzie wykorzystywana jest metoda gradientu prostego uczenie opiera się na dwu głównych krokach:

- propagacja sygnału w przód - dane przechodzą przez kolejne warstwy neuronów i na podstawie wag i funkcji aktywacji na wyjściu sieci generowany jest wynik,
- propagacja błędu wstecz - obliczony błąd między wynikami (najczęściej MSE) jest propagowany wstecz przez sieć, podczas której obliczane są gradienty funkcji błędu względem wag sieci w celu aktualizacji wag, tak by zminimalizować funkcję błędu.

Takie rozwiązanie wiąże się jednak z potencjalnymi problemami, które możemy wyeliminować implementując do celów uczenia sieci neuronowych algorytmy ewolucyjne. Takie potencjalne problemy to między innymi:

- Zatrzymywanie się w ekstremach lokalnych:
    - Algorytmy ewolucyjne mogą lepiej radzić sobie z poszukiwaniem globalnych minimów funkcji błędu, zwłaszcza w trudnych przestrzeniach rozwiązań (np. zaszumionych, wielomodalnych) co czasem może być problematyczne dla metod gradientowych.
- Odporność na znikający gradient:
    - W przeciwności do metod gradientowych, algorytmy ewolucyjne nie wykorzystują gradientów - nie są więc podatne na zjawisko znikającego gradientu które może wystąpić w głębokich sieciach neuronowych, gdzie gradient może maleć lub rosnąć eksponencjalnie.

Należy jednak pamiętać, że implementacja algorytmów ewolucyjnych do tego typu zadania również wiąże się z pewnymi problemami. Problemy te występują głównie w sferze wydajności i czasu uczenia modeli, co jest spowodowane większą zasobochłonnością obliczeniową w stosunku do metod gradientowych i jest szczególnie widoczne w przypadku dużych zbiorów danych i skomplikowanych modeli.

2. Omówienie zaimplementowanego rozwiązania

Zagadnienie uczenia sieci neuronowych algorytmami ewolucyjnymi jest złożonym i dość niszowym tematem, w celu realizacji tego zadania w niektórych kwestiach wspieraliśmy się artykułem z Mathematics Journal - Differential Evolution for Neural Networks Optimization.

Zadanie uczenia sieci neuronowej algorytmami ewolucyjnymi zdefiniowaliśmy jako taką optymalizację wag sieci, by generowane przewidywania były jak najbardziej dokładne.

Osobnikiem w stosowanych algorytmach ewolucyjnych jest zlinearyzowany wektor wag:
$[W(1), \ldots, W(K)]$
Gdzie kolejne wektory $W(k)$ reprezentują wagi odpowiadających warstw sieci neuronowej, $k = 1, \ldots, K$

Optymalizacja sieci polega na minimalizacji funkcji dopasowania f osobnika. Funkcja dopasowania dla danego osobnika wyznaczana jest na podstawie predykcji modelu (z wagami przypisanymi z osobnika) na zbiorze walidacyjnym.

Funkcja dopasowania f zdefiniowana jako entropia krzyżowa - liczona przy użyciu `keras.losses.categorical_crossentropy`.
Alternatywnie jako funkcja dopasowania testowana była dokładność modelu (accuracy) - liczona przy użyciu `sklearn.metrics.accuracy_score`. Dla niedużej liczby generacji (<100) uzyskiwane wyniki były podobne.

Wybrane metody optymalizacji:

1) Ewolucja różnicowa w wariancie **DE/best/1/bin**
   Implementowana przez funkcję *de_best_1_bin_evolve* z pliku algorithms.py
   Wykorzystujemy wersję algorytmu ewolucji różnicowej z selekcją punktu roboczego jako najlepszego ze zbioru, jedną parą punktów do mutacji wybieranych losowo i wybór najlepszego osobnika z 2-elementowej selekcji turniejowej.

   Ogólna struktura działania algorytmu wygląda następująco:

Algorytm Ewolucji Różnicowej		
1.	Inicjuj $P^0 \leftarrow \{x_1, x_2 \ldots x_\mu\}$	Inicjalizacja populacji początkowej P^0 μ osobnikami
2.	$H \leftarrow P^0$	Inicjalizacja historii H populacją początkową
3.	$t \leftarrow 0$	
4.	while !stop	
5.	for ($i \in 1{:}\mu$)	Dla każdego i-tego punktu w populacji:
6.	$r_i^t \leftarrow select(P^t)$	Wybór punktu roboczego r_i^t
7.	$d_i^t, e_i^t \leftarrow sample(P^t, 2)$	Wybór dwóch losowych punktów d_j^t i e_k^t na podstawie których będzie dokonana modyfikacja
8.	$M_i^t \leftarrow r_i^t + F(e_i^t - d_i^t)$	Nowy punkt M_i^t jest modyfikacją roboczego na podstawie różnicy między dwoma wylosowanymi
9.	$O_i^t \leftarrow crossover(r_i^t, M_i^t)$	Krzyżowanie zmodyfikowanego punktu z oryginalnym
10.	$H \leftarrow H \cup \{O_i^t\}$	Dodanie punktu do historii
11.	$P_i^{t+1} \leftarrow tournament(p_i^t, O_i^t)$	Aktualizacja populacji: podmiana i-tego punktu jeżeli nowy jest lepszy
12.	$t \leftarrow t + 1$	

   W szczególności algorytm implementuje:
   - wybór punktu roboczego - osobnik z populacji o najlepszym dopasowaniu
   - wybór punktów do mutacji - dwa różne losowo wybrane osobniki
   - mutacja - dodanie do wag punktu roboczego różnicy wylosowanych punktów pomnożona o współczynnik F
   - krzyżowanie - binarne, źródło każdej wagi losowane niezależnie, gdzie współczynnik Cr z przedziału (0, 1) mówi z jakim dużym prawdopodobieństwem waga będzie przypisana z punktu roboczego
   - selekcja turniejowa - wybierany lepszy osobnik z pary: punkt wynikowy krzyżowania, i-ty punkt w populacji

   W tej wersji współczynniki F, oraz Cr przypisywane są z góry, co sprawia, że algorytm ma mniejsze zdolności dopasowywania się do aktualnego położenia. Podjęliśmy próby implementacji wersji JADE, w którym współczynniki te są dynamicznie modyfikowane, natomiast prawdopodobnie w wyniku nieoptymalnej implementacji nie wpłynęło to pozytywnie na działanie algorytmu.

2) Strategia ewolucyjna **(μ+λ)-ES**
   Implementowana przez funkcję *mu_lambda_es_evolve* z pliku algorithms.py
   Jedna z podstawowych ewolucyjnych używanych do optymalizacji, o względnie prostym sposobie działania. Składa się z dwóch kluczowych parametrów: μ (liczba rodziców) i λ (liczba potomków). Algorytm działa według następujących kroków:

   Inicjalizacja populacji:
   Losowo incjalizowanych jest μ osobników, które stanowią populację początkową.

   Dla kolejnych t generacji:

   - **Generacja potomków:**
     i. Dla każdego z μ rodziców generowanych jest λ potomków poprzez dodanie do wag losowego szumu z rozkładu normalnego o średniej 0 i odchyleniu standardowym 1.
   - **Ewaluacja:**
     i. Dla każdego rodzica i potomka obliczana jest wartość funkcji dopasowania.
   - **Selekcja:**
     i. Populacja początkowa (rodziców) łączona jest z wygenerowanymi potomkami, a następnie spośród nich wybieranych jest μ najlepszych osobników.
   - **Aktualizacja populacji:**
     i. Nowa populacja tworzona jest z μ osobników, które zostały wybrane w poprzednim kroku.

3) Metoda **stochastycznego spadku gradientu**:
   Implementowana, jak opisano wyżej, na zasadzie feedforwardingu i backpropagation. W wersji stochastycznej do obliczenia gradientu funkcji błędu w danym kroku wykorzystywana jest tylko losowo wybrana część zbiór treningowego, a nie całość, jak w przypadku gradientu prostego.

   W kodzie wykorzystywany jest optimizer sgd z biblioteki tensorflow.

3. Różnice względem wstępnej wersji dokumentacji

- Zrezygnowaliśmy z wykorzystania algorytmu CMA-ES, oraz jego wersji pochodnych, ze względu na ich dużą złożoność, oraz trudności w implementacji. Zamiast tego korzystamy ze strategii ewolucyjnej (μ+λ)-ES
- Wybrana metoda gradientowa to SGD, zamiast wstępnie wybranej metody gradientu prostego.
- Zamiast wstępnie zakładanej ewolucji różnicowej SHADE wykorzystana została wersja ze stałymi współczynnikami F i Cr - DE/best/1/bin
- Dla testów zdecydowaliśmy się skorzystać tylko z architektur FNN i CNN, pozostałe dwie architektury okazały się na tyle złożone, że przeliczenie nawet niewielkiej liczby generacji trwało bardzo długo.

4.1 Eksperymenty - założenia

Celem przeprowadzanych eksperymentów jest porównanie działania wytrenowanych modeli sieci neuronowych z wykorzystaniem różnych metod uczenia. Analizować będziemy jakość działania sieci w kontekście zadania klasyfikacji, główne miary które będziemy wyznaczać dla modeli to: dokładność (accuracy), macierz pomyłek (confusion matrix), precyzja (precision) i czułość (recall).

Eksperymenty będą przeprowadzone w 3 krokach:

1. *Inicjalizacja modelu:* utworzenie modelu nowego modelu sieci neuronowej
2. *Uczenie modelu:* przeprowadzanie treningu sieci wybraną metodą
3. *Ewaluacja modelu :* predykcja na zbiorze testowym, obliczenie miar jakości

Do przeprowadzenia eksperymentów wykorzystywany jest zbiór mnist jako standardowy zbiór w kontekście problemu klasyfikacji.

Testowane architektury:
Prosta Sieć Neuronowa (FNN):

- Warstwa wejściowa: (28x28 pikseli obrazu MNIST)
- Warstwa spłaszczająca
- Warstwa ukryta: 128 neuronów, funkcja aktywacji: ReLU
- Warstwa wyjściowa: 10 neuronów, funkcja aktywacji: Softmax

Konwolucyjna Sieć Neuronowa (CNN):

- Warstwa wejściowa: 28x28x1 (1 kanał obrazu)
- Konwolucyjna warstwa: 32 filtry, rozmiar jądra 3x3, funkcja aktywacji ReLU
- Max Pooling: Rozmiar 2x2
- Konwolucyjna warstwa: 64 filtry, rozmiar jądra 3x3, funkcja aktywacji ReLU
- Max Pooling: Rozmiar 2x2
- Warstwa spłaszczająca
- Warstwa ukryta: 128 neuronów, funkcja aktywacji ReLU
- Warstwa wyjściowa: 10 neuronów, funkcja aktywacji Softmax

4.2 Eksperymenty - uzyskane wyniki

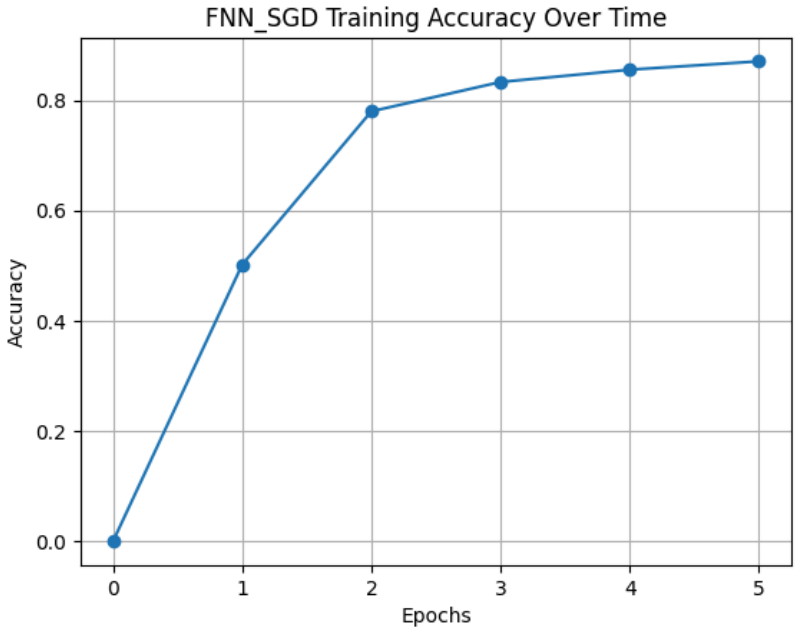
Powtarzalność wyników jest zapewniana dzięki zastosowaniu funkcji *set_seed()*, która ustawia jednakowe ziarno generatora dla wszystkich wykorzystywanych bibliotek.

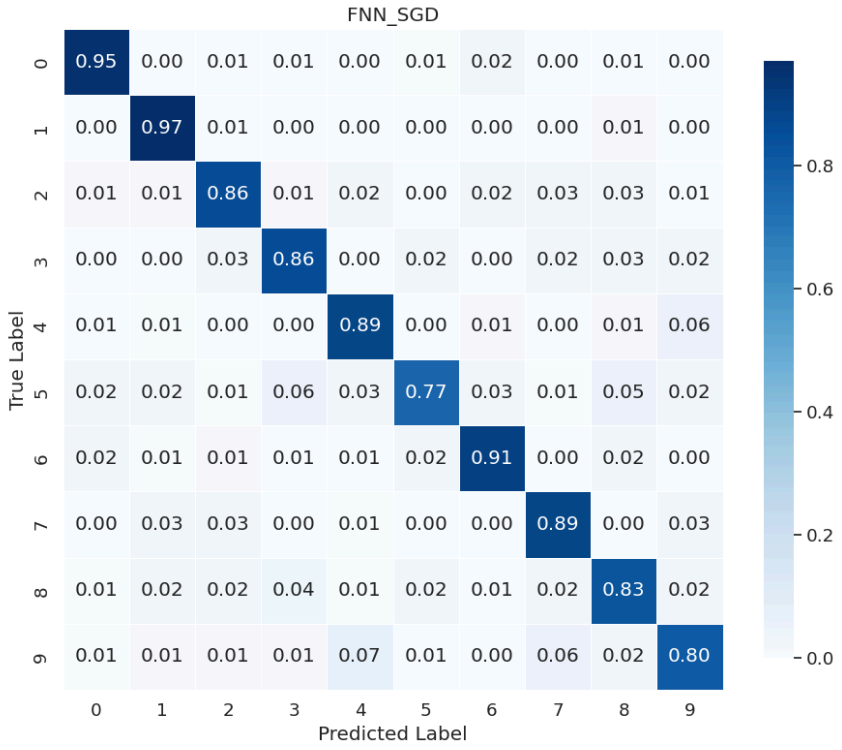
Eksperymenty uruchomić można przy użyciu notatnika *experiments.ipynb*.

Poniżej dla każdej sieci i metody optymalizacji znajdują się wykresy i miary opisujące jakość modelu i przebieg jego treningu. W szczególności:

- Wykres accuracy w zależności od generacji modelu
- Macierz pomyłek dla przewidywanych klas
- Miary jakości liczone z użyciem biblioteki sklearn:
    - accuracy, precision, recall
- Czas wykonania

Sieć FNN - optymalizator SGD

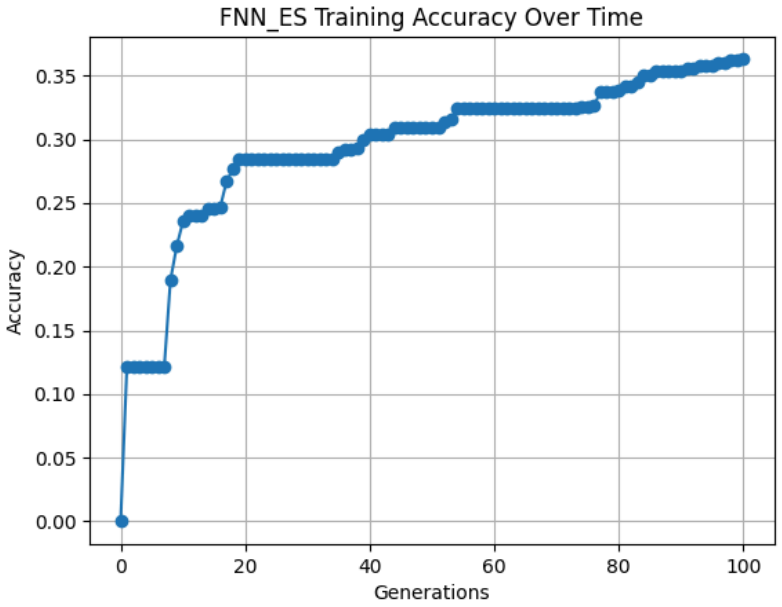


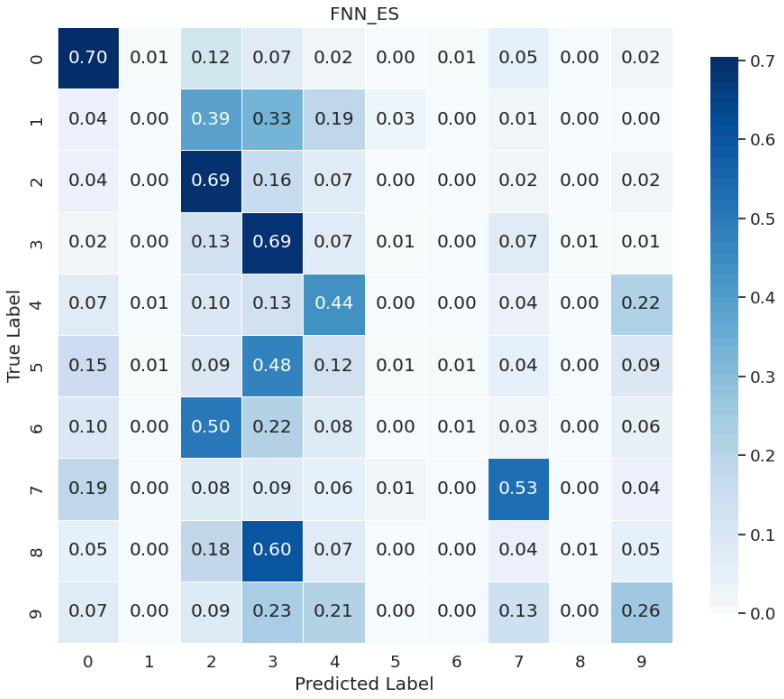


Miary jakości modelu:

- Accuracy: 0.8752
- Precision: 0.8750
- Recall: 0.8752
- Czas wykonania: 3.12 s

Sieć FNN - mu+lambda, gdzie mu=5, lambda=5, liczba generacji = 100

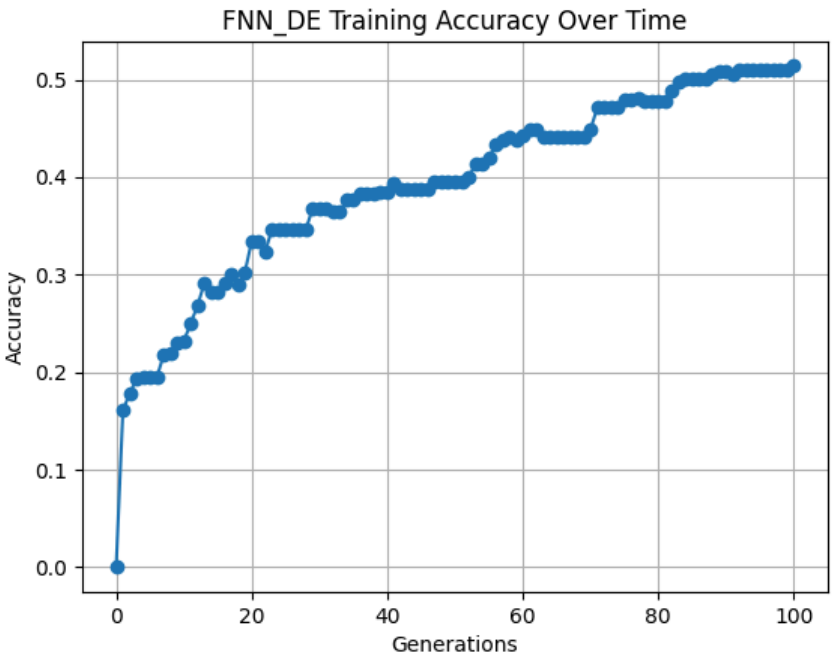


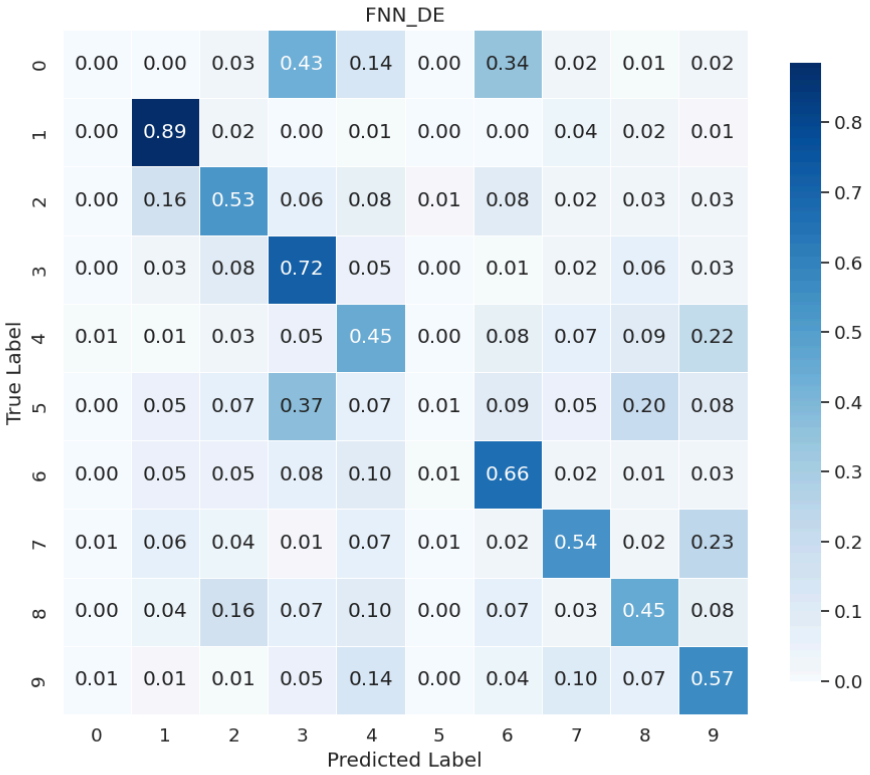


Miary jakości modelu:

- Accuracy: 0.3344
- Precision: 0.2732
- Recall: 0.3344
- Czas wykonania: 431.93 s

Sieć FNN - DE, gdzie F=1, Cr=0.2, populacja=10, liczba generacji = 100

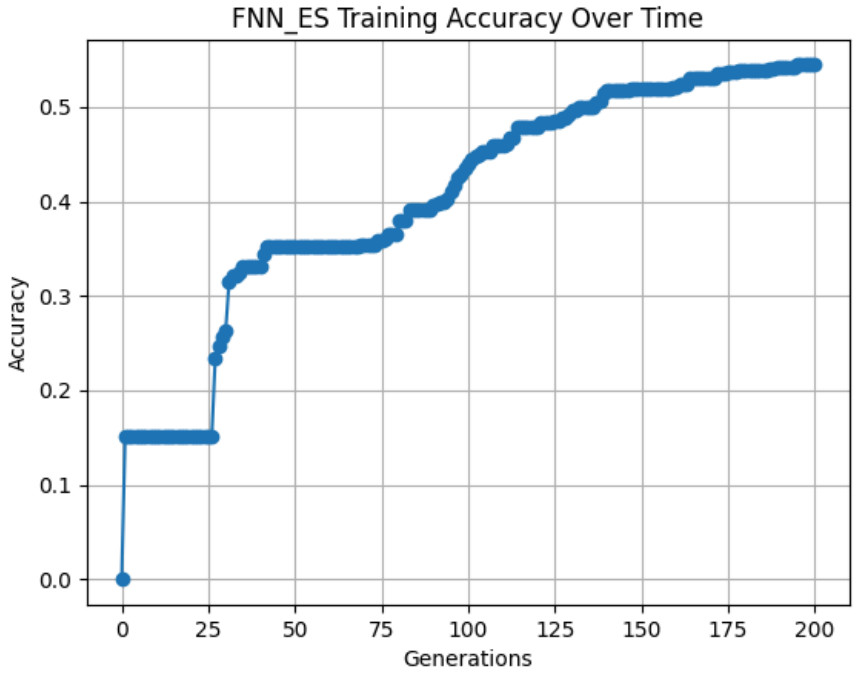


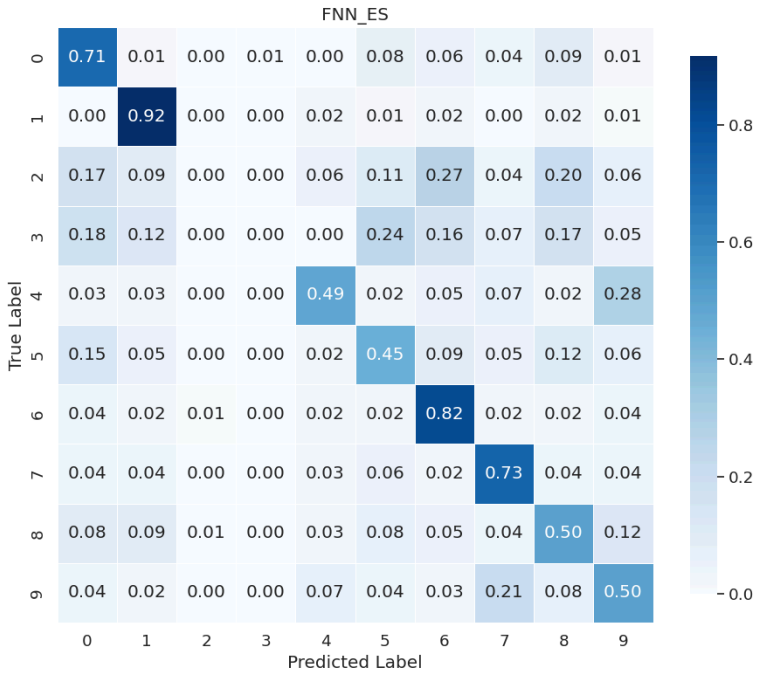


Miary jakości modelu:

- Accuracy: 0.4932
- Precision: 0.4251
- Recall: 0.4932
- Czas wykonania: 429.76 s

Sieć FNN - mu+lambda, gdzie mu=5, lambda=5, liczba generacji = 200

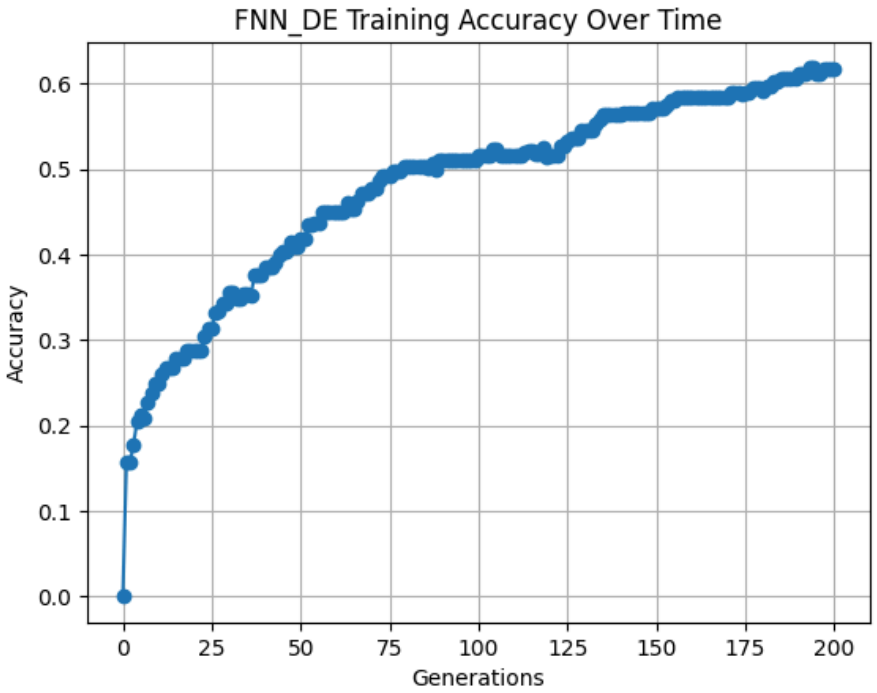


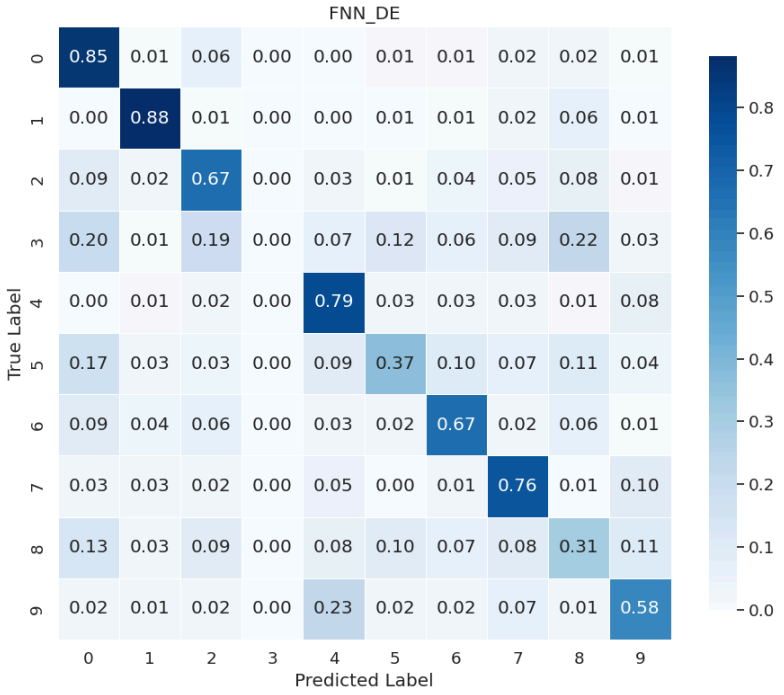


Miary jakości modelu:

- Accuracy: 0.5153
- Precision: 0.4266
- Recall: 0.5153
- Czas wykonania: 1788.83 s

Sieć FNN - DE, gdzie F=1, Cr=0.2, populacja=20, liczba generacji = 200

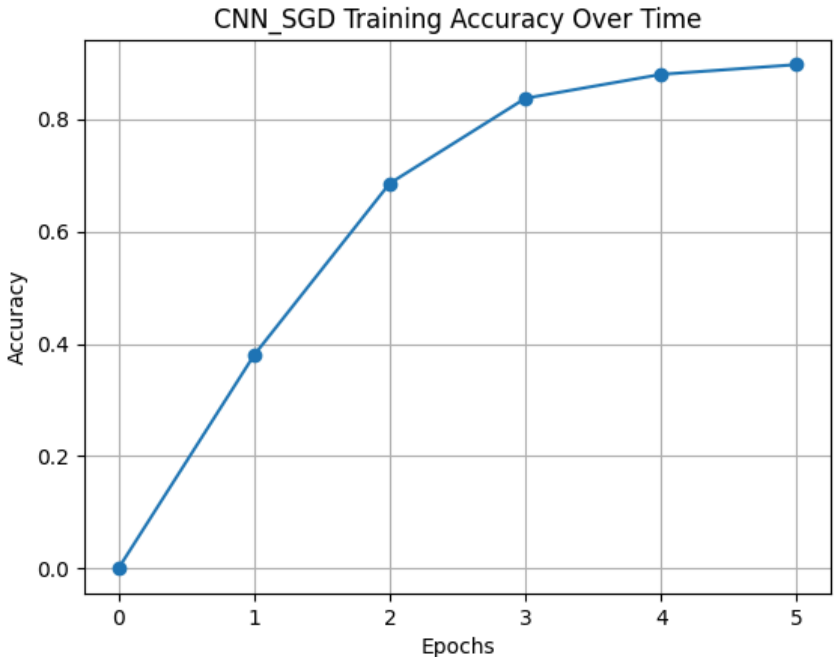


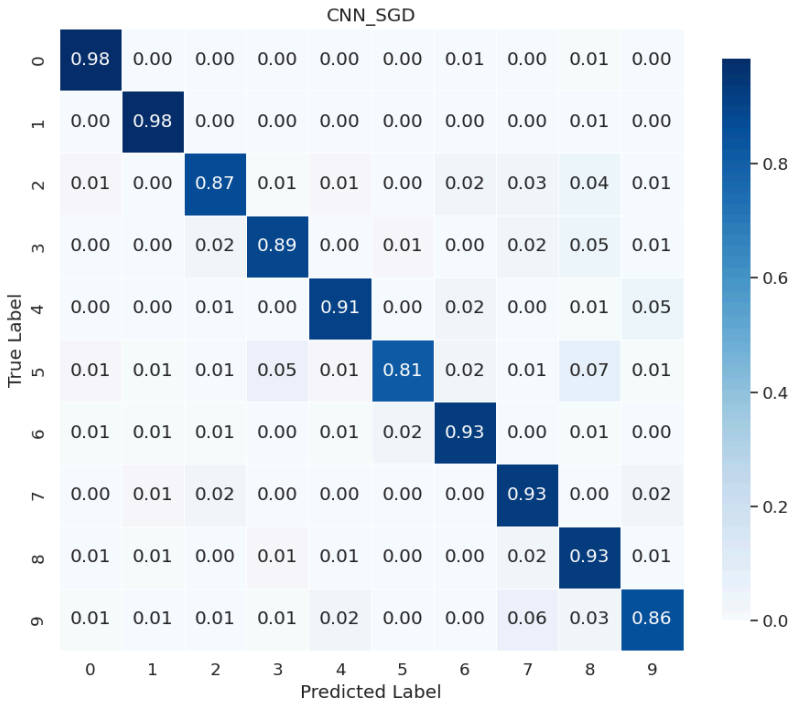


Miary jakości modelu:

- Accuracy: 0.5941
- Precision: 0.5315
- Recall: 0.5941
- Czas wykonania: 931.39 s

Sieć CNN - optymalizator SGD

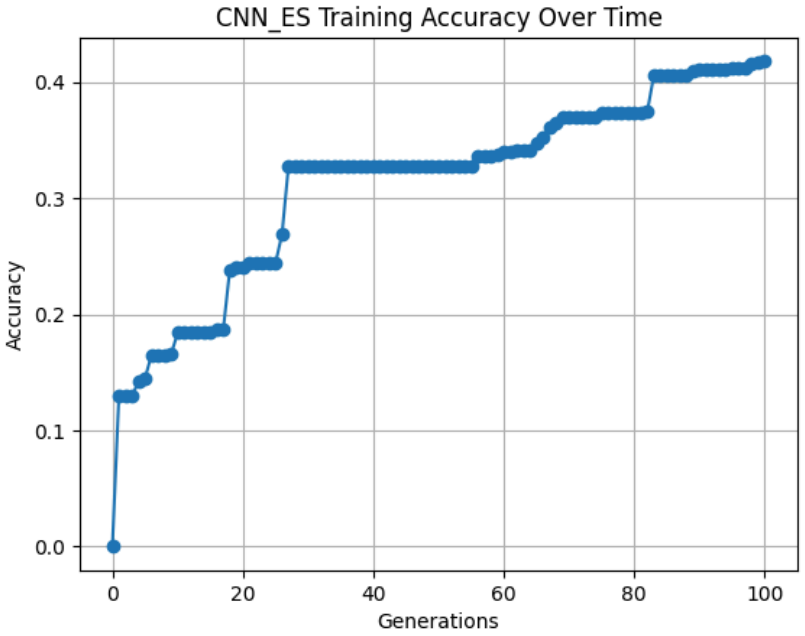


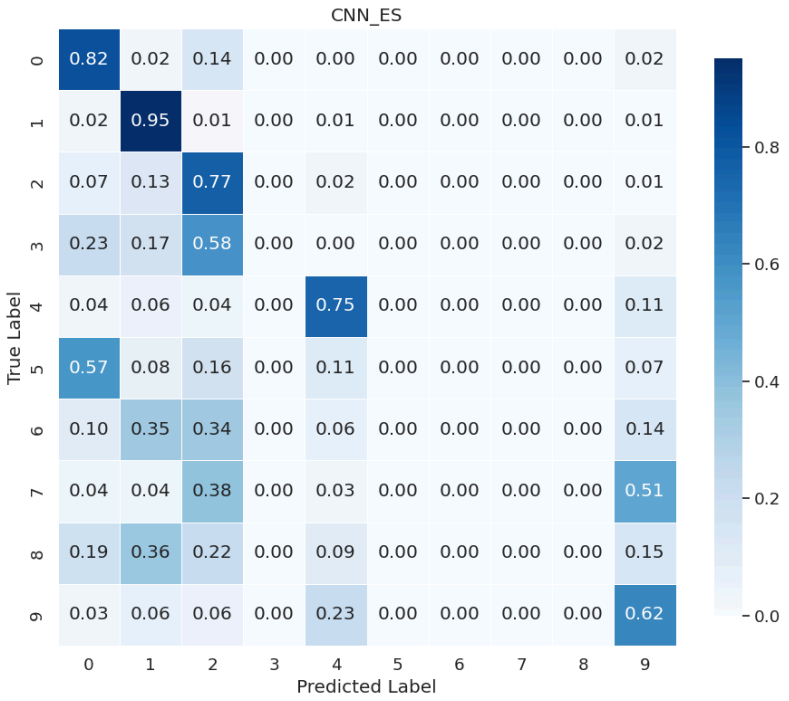


Miary jakości modelu:

- Accuracy: 0.9119
- Precision: 0.9141
- Recall: 0.9119
- Czas wykonania: 20.32 s

Sieć CNN - mu+lambda, gdzie mu=5, lambda=5, liczba generacji = 100

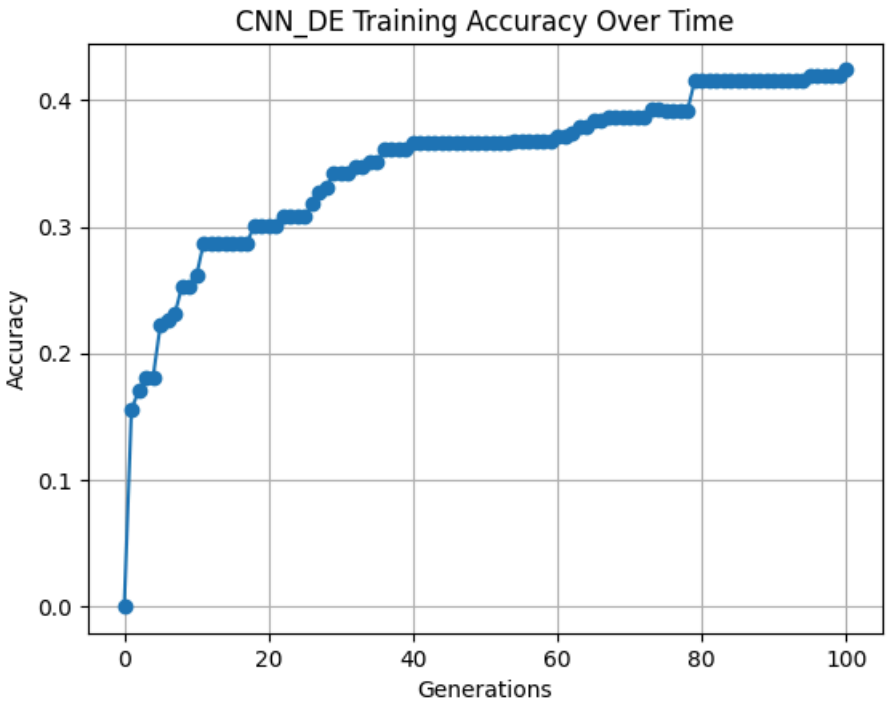


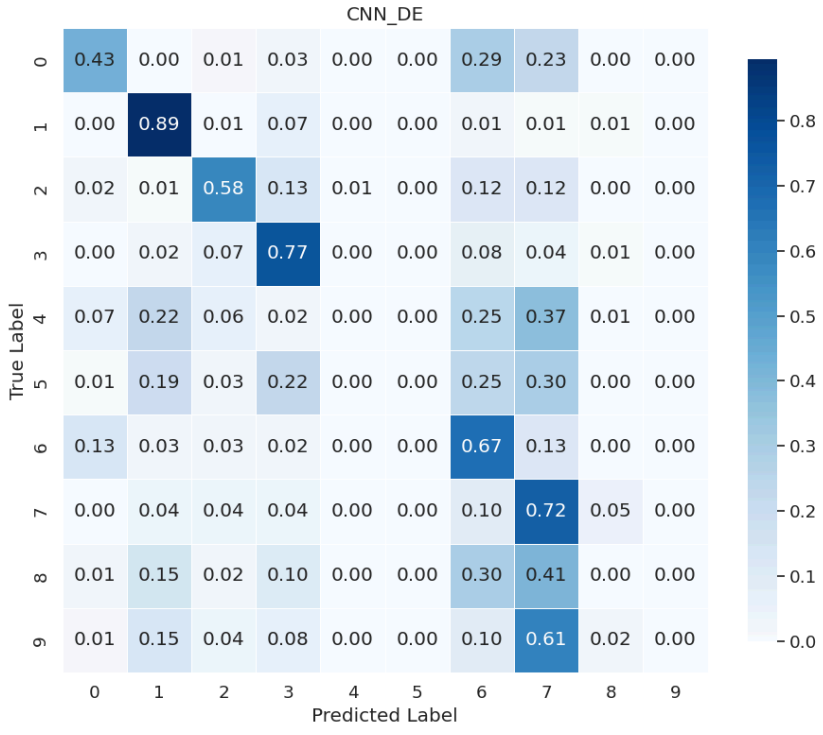


Miary jakości modelu:

- Accuracy: 0.4043
- Precision: 0.2169
- Recall: 0.4043
- Czas wykonania: 1414.41 s

Sieć CNN - DE, gdzie F=1, Cr=0.2, populacja=10, liczba generacji = 100





Miary jakości modelu:

- Accuracy: 0.4191
- Precision: 0.3389
- Recall: 0.4191
- Czas wykonania: 1360.50 s

5. Omówienie wyników i wnioski

Uzyskane wyniki pokazują, że algorytmy ewolucyjne wykonują się zdecydowanie dłużej. Nie skutkuje to jednak dostatecznie pozytywnymi wynikami. Po 100/200 generacjach wyniki dla obu algorytmów ewolucyjnych są znacznie gorsze od wyników uzyskiwanych przez model trenowany przy użyciu optymalizatora SGD z biblioteki tensorflow.

Można zaobserwować, że algorytmy ewolucyjne czasami przez kilka generacji nie były w stanie znaleźć lepszego punktu. Jest to jednak spodziewane zachowanie z uwagi na rozmiar populacji. Mimo tego, jakość modelu powoli poprawia się w kolejnych generacjach.

Podsumowanie
Użycie strategii ewolucyjnych w kontekście uczenia sieci neuronowych ma spory potencjał ze względu na możliwość przeszukiwania w całym obszarze rozwiązań.
W idealnej sytuacji pozwoliłoby to na znalezienie wag modelu o lepszej dokładności niż te wyliczone na podstawie gradientu.

Problemem jest jednak wysoka złożoność obliczeniowa i przestrzeń przeszukiwań. Dla zwykłego modelu FNN ilość wszystkich wag to aż 101770 - aby móc uzyskać prawdziwie dobre wyniki, ilości osobników w populacji powinna być stosownie duża. Niestety są to obliczenia niemożliwe do wykonania na zwykłym komputerze.

Oczywiście, należy zwrócić uwagę na optymalność naszego rozwiązania, w którym jest wiele miejsc na możliwe poprawki, które mogą pozytywnie wpłynąć na czas wykonania algorytmów. (batching zbioru walidacyjnego, usprawnienie krzyżowania i mutacji, refactoring kodu, tak by możliwie jak najmniej działań było wykonywane w pętli)

Z tych powodów praktyczna użyteczność takich algorytmów dla zwykłego użytkownika jest niewielka, szczególnie w porównaniu do szybkości trenowania i jakości modelu uzyskiwanego przy wykorzystaniu metod gradientowych.

6. Użyte technologie

Projekt zdecydowaliśmy się realizować w języku **Python**, głównie ze względu na dużą dostępność narzędzi i bibliotek ułatwiających realizację zagadnień z domeny Machine Learning.
Stack technologiczny:

- *Jupyter Notebook* - przeprowadzenie eksperymentów, analiza otrzymanych wyników
- *NumPy* - operacje numeryczne i manipulacja danymi
- *pandas* - zarządzanie, manipulacja zbiorami danych
- *DEAP* - implementacja algorytmów ewolucyjnych
- *Keras (TensorFlow)* - implementacja sieci neuronowych, pobranie datasetu MNIST, enkodowanie danych dyskretnych (labeli)
- *scikit-learn* - tworzenie zbioru walidacyjnego, mierzenie metryk, tworzenie macierzy pomyłek