Sprawozdanie 2
Sieci Kohonena
MIOwAD

Mateusz Nizwantowski
01161932@pw.edu.pl
313839

12 maja 2024

Spis treści

1 Wstęp

W ramach przedmiotu Metody Inżynierii Obliczeniowej w Analizie Danych poznaliśmy temat sieci Kohonena. Dla wielu z nas było to pierwsze spotkanie z tą tematyką. Celem jest implementacja from scratch podstawowej sieci Kohonena znanej też pod nazwą Self-organizing map lub w skrócie SOM. Tym razem podobnie jak w przypadku sieci neuronowych co tydzień mieliśmy do wykonania etapy, które rozbudowywały dotychczasowe rozwiązanie, jednak tym razem były to tylko 2 etapy. Dodatkowo testowaliśmy naszą sieć na różnorodnych problemach.
Na końcu naszym finalnym zadaniem jest opisanie naszych doświadczeń w postaci raportu. Przedstawienie co udało nam się osiągnąć, jakie problemy napotkaliśmy, czego się nauczyliśmy i w jaki sposób planujemy rozwijać wiedzę zdobytą podczas tego etapu. Cały kod, który napisałem w ramach tego projektu znajduje się w repozytorium na GitHub.

2 Teoria

Sieć Kohonena, znana również jako mapa Kohonena, jest rodzajem sieci neuronowej używanej w problemach uczenia nienadzorowanego. Została zaproponowana przez fińskiego matematyka Teuvo Kohonena w latach 80. XX wieku. Sieć Kohonena jest szczególnym przypadkiem samo-organizujących się map, które są zdolne do grupowania danych i zachowywania ich topologicznej struktury. Jest to przykład sieci konkurencyjnej, a więc takiej, w której sygnały wyjściowe neuronów porównuje się ze sobą w celu wskazania zwycięzcy (zwycięski neuron może np. wskazywać klasyfikację sygnału wejściowego). Sieć wykorzystuje koncepcję sąsiedztwa. W wyniku uczenia tej sieci powstaje mapa topologiczna, w której neurony reprezentujące podobne klasy powinny znajdować się blisko siebie. Dzięki temu możliwe jest zaobserwowanie pewnych relacji pomiędzy klasami. Na podstawie analizy konkretnych przykładów danych wejściowych zazwyczaj można ustalić, jakie znaczenie mają poszczególne rejony tej mapy.

2.1 Architektura sieci Kohonena

- Sieć Kohonena składa się z warstwy neuronów wejściowych oraz warstwy konkurencyjnej, zwanej również jako warstwa topologiczna lub warstwa Kohonena. Warstwa wejściowa nie dokonuje żadnych przekształceń danych, odpowiada ona tylko za rozprowadzenie danych wejściowych do neuronów warstwy konkurencyjnej.
- Neurony wejściowe są połączone z neuronami Kohonena poprzez wagi, które ulegają modyfikacji w procesie uczenia.
- Neurony Kohonena są ułożone w dwuwymiarową lub wielowymiarową siatkę, co umożliwia zachowanie topologii danych. Topologia sieci (ułożenie neuronów) nie ulega zmianie podczas procesu uczenia.

2.2 Sposób uczenia się sieci Kohonena

- Na początku losowo inicjalizacje się wagi dla każdego neuronu z warstwy Kohonena
- Podczas procesu uczenia, dla każdej prezentowanej próbki danych, sieć wybiera neuron Kohonena, który jest najbardziej zbliżony do wektora wejściowego. Ten proces nazywany jest konkurencją. Podobieństwo określa się za pomocą jakiejś metryki, najczęściej euklidesowej.
- Następnie wagi neuronów Kohonena są aktualizowane w celu zbliżenia się do wektora wejściowego. Neuron, który był zwycięzcą, oraz neurony sąsiednie, otrzymują większe korekty wag niż neurony bardziej oddalone.
- Proces ten sprawia, że neurony Kohonena uczą się reprezentować strukturę danych poprzez ułożenie się w podobne regiony przestrzeni, co prowadzi do mapowania danych.
- Jest to proces iteracyjny: sieci pokazuje się wszystkie próbki w naszym zbiorze, a i sam zbiór pokazuje się kilka razy, tak żeby sieć miała okazje dostosować się do topologii danych

2.3 Reguła Hebba w uczeniu sieci Kohonena

- Reguła Hebba, znana również jako zasada Hebba, jest podstawową zasadą uczenia się w sieciach neuronowych, która mówi, że neurony, które wykazują aktywność jednocześnie, wzmacniają swoje połączenia.
- W kontekście sieci Kohonena, reguła Hebba jest wykorzystywana do aktualizacji wag po wyborze zwycięzcy konkurencji.

Sieci Kohonena znajdują zastosowanie w wielu dziedzinach, takich jak analiza danych, rozpoznawanie wzorców, segmentacja obrazów czy klastrowanie danych. Ich zdolność do samoorganizacji i zachowania topologicznej struktury danych czyni je użytecznym narzędziem w analizie i wizualizacji złożonych zbiorów danych.

3 Eksperymenty

3.1 KOH1

Jak wcześniej pisałem w tym raporcie są tylko 2 etapy. Na zrobienie pierwszego mieliśmy aż 2 tygodnie. Jego celem było zaimplementować podstawową sieć Kohonena z prostokątną siatką (sieć należało zaprogramować tak, aby można było jako parametry podać rozmiary siatki. Dodatkowo trzeba było napisać 2 funkcje sąsiedztwa z możliwością zmiany szerokości sąsiedztwa:

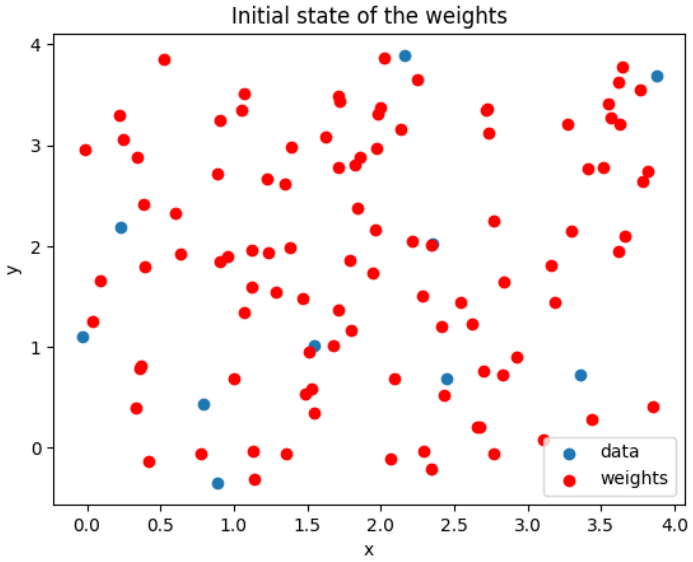
- funkcję gaussowską,
- minus drugą pochodną funkcji gaussowskiej znanej też pod nazwą meksykański kapelusz.

Jako funkcję wygaszającą należało użyć funkcji z wykładniczym wygaszaniem. Część implementacyjna była bardzo prosta i dało się ją zrobić przepisując pseudokod ze slajdu z wykładu. Nie miałem z nią większych problemów poza tym, że na slajdach jest błąd w jednym miejscu, jednak dało się do tego dojść "na logikę".

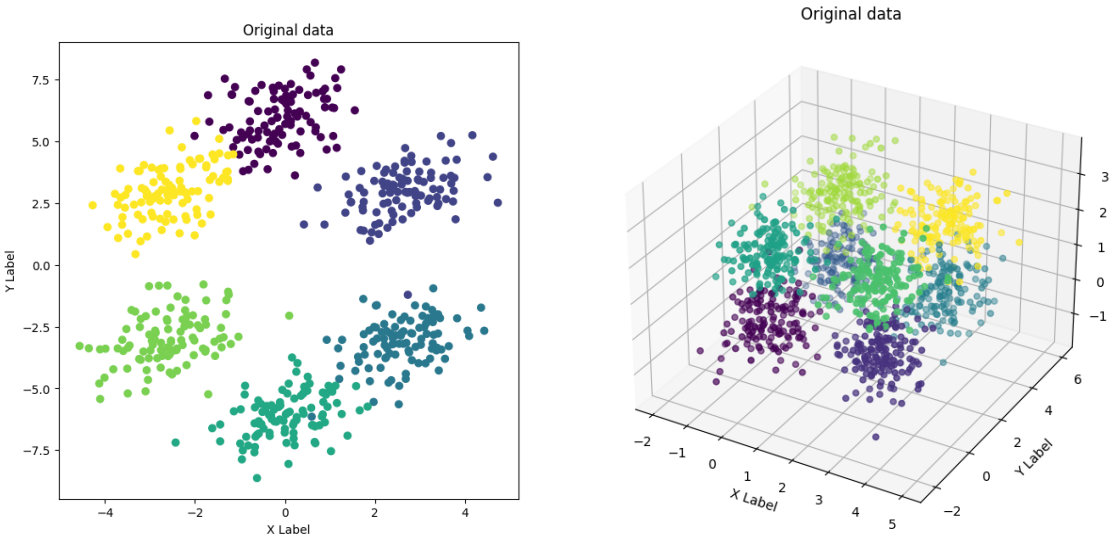
Moją sieć inicjalizacje następująco:

- w każdym wymiarze danych wejściowych znajduję wartość maksymalną i minimalną
- dla każdego neuronu i dla każdego wymiaru losuję liczbę z rozkładu jednostajnego [0,1]
- dokonuję odpowiedniego skalowania tak aby wagi w danym wymiarze zawierały się od wartości minimalnej do maksymalnej

Wygląda to następująco: punkty niebieskie to dane wejściowe a czerwone to wagi neuronów.



Następnie należało przetestować napisaną implementację na 2 podanych zbiorach: hexagon i cube.

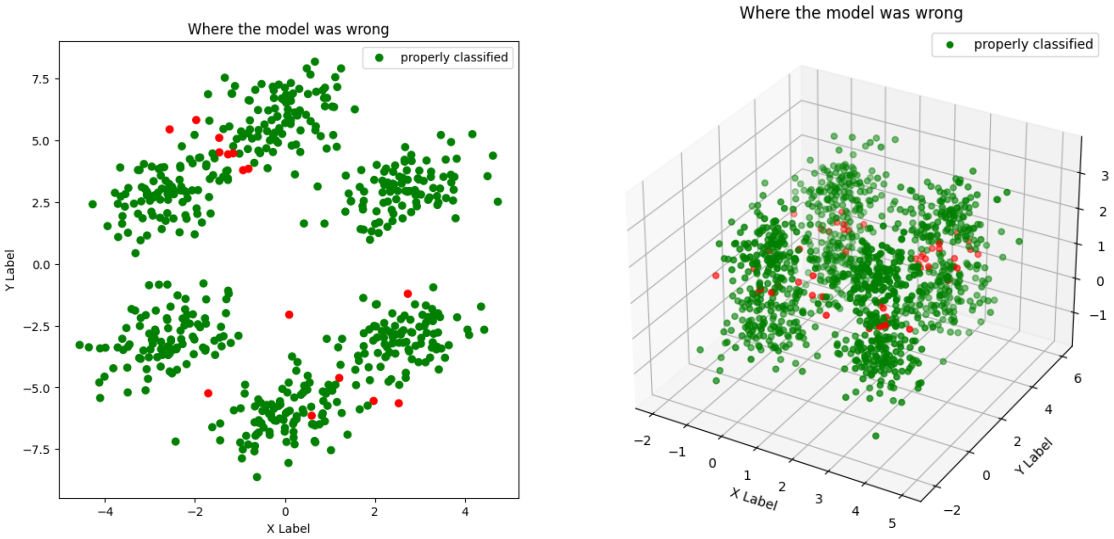


(a) dane hexagon (b) dane cube

Rysunek 1: Rozkład danych cube i hexagon

Jak widać są to dane sztucznie wygenerowane, relatywnie dobrze separowalne, więc nie powinny być dla sieci Kohonena problemem. Co ważne w testowaniu implementacji są to też dane nisko wymiarowe, które jestem w stanie zwizualizować. Pokazać gdzie, model się pomylił jak przypisał klastry itp. Jest to cenna własność przy ewentualnym debuggowaniu, które na szczęście nie miało miejsca w tej implementacji. Dodatkowo będę używał miary v-measure. Mówiąc w prostych słowach jest to odpowiednik f-score z zadania klasyfikacji.

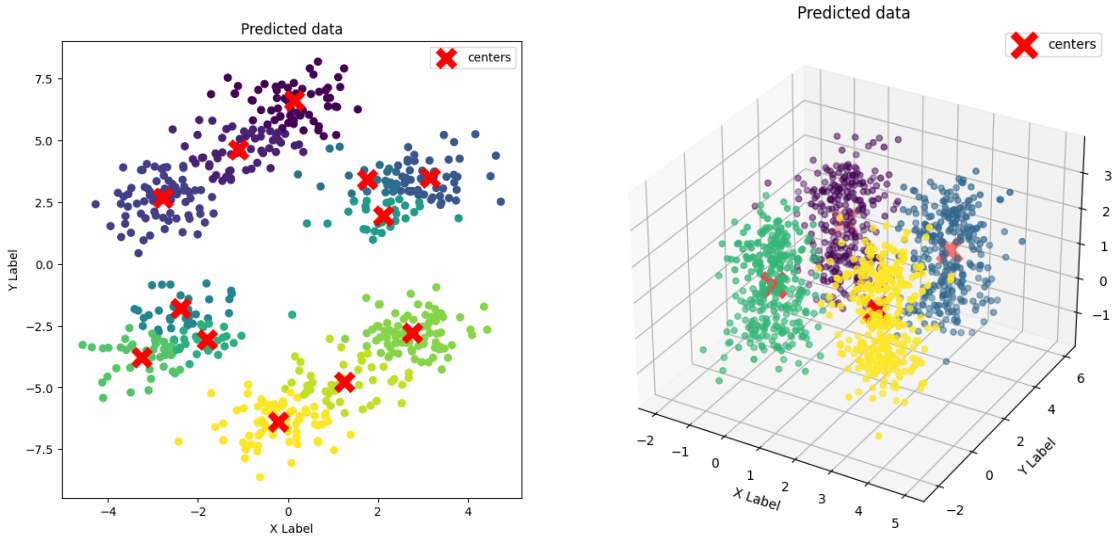
Tak jak przewidziałem te problemy nie stanowią trudności dla sieci Kohonena w obu przypadkach uzyskujemy v-measure większe niż 0.92. Jak widać na wykresach poniżej błędy popełniamy na granicach gdzie nie ma jasnej granicy decyzyjnej, co jest zgodne z oczekiwaniami. Zielone kropki to odpowiednio sklastrowane obserwacje, czerwone błędne przyporządkowane.



(a) dane hexagon (b) dane cube

Rysunek 2: Gdzie popełniłem błędy

Wyniki nie są tak dobre kiedy weźmiemy inny rozmiar siatki neuronów a co za tym idzie inną liczbę klastrów. W zadaniu klastryzacji, brak znajomości a priori liczby klastrów jest typową sytuacją. Jednak pomimo źle dobranej liczby klastrów dane cube i hexagon zachowują się relatywnie dobrze. Poniżej przedstawione są przykładowe przyporządkowania dla danych hexagon ze zbyt dużą liczbą zdefiniowanych klastrów i dane cube z za małą zdefiniowaną liczbą klastrów. Ciekawe spostrzeżenia jakie możemy wysnuć z tych wizualizacji to fakt, że przy zmniejszeniu liczby klastrów model skleił klastry, z którymi miał poprzednio problemy. Są to przypadki w których nie ma jasnej granicy decyzyjnej. Nie powiedziałbym, że uzyskane wyniki są bezużyteczne, musimy pamiętać, że rozpatrywany przypadek jest nierealistyczny. Normalnie nie znamy "prawidłowego" przyporządkowania. Oba wyniki niosą jakąś skondensowaną informacje o danych, jednak my znając prawidłowe etykiety wiemy, że da się lepiej. Nie mniej jednak, gdyby ktoś pokazał mi dane cube bez żadnego kontekstu i kazał pokazać klastry zrobił bym to identycznie jak model poniżej, wydaje się to być najnaturalniejsza rzecz do zrobienia. Na danych hexagon poniższe przyporządkowanie skutkuje v-measure na poziomie 0.78, identyczny wynik uzyskujemy dla danych cube - v-mesure = 0.78. Na czerwono zaznaczone są wagi neuronów z warstwy Kohonena, można je interpretować jako środki klastrów.

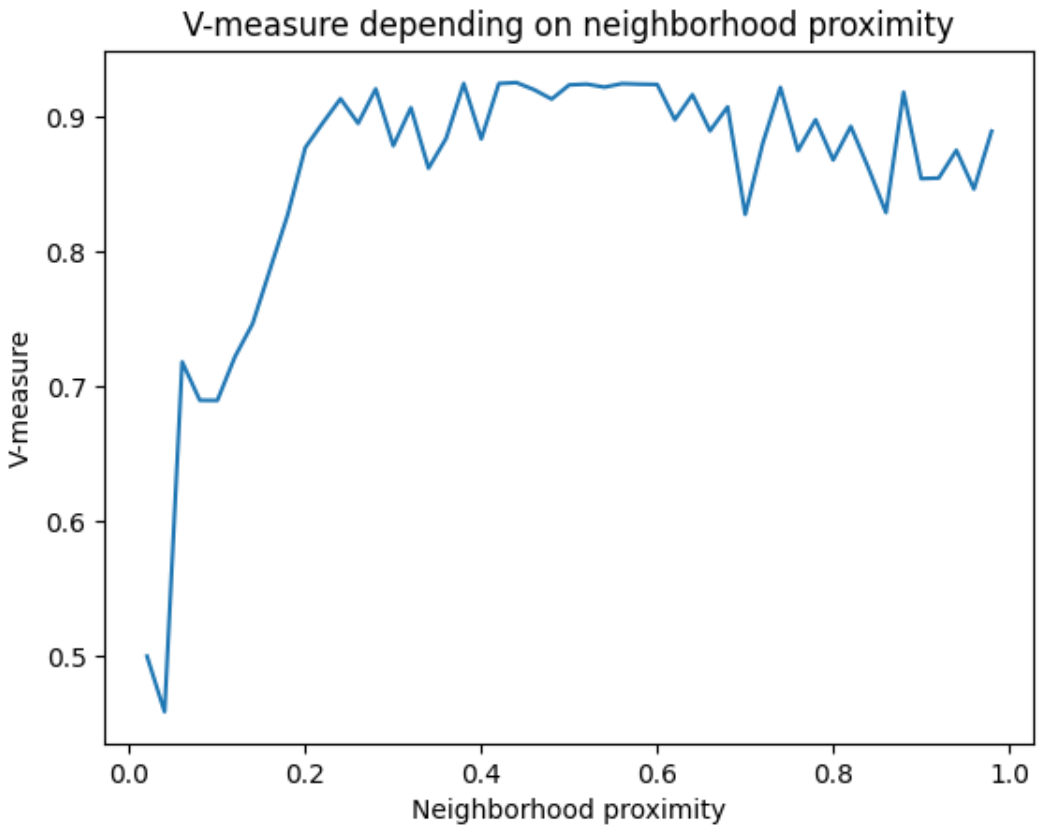


(a) dane hexagon

(b) dane cube

Rysunek 3: Jak zachowuje się model w obliczu niepoprawnej liczby klastrów

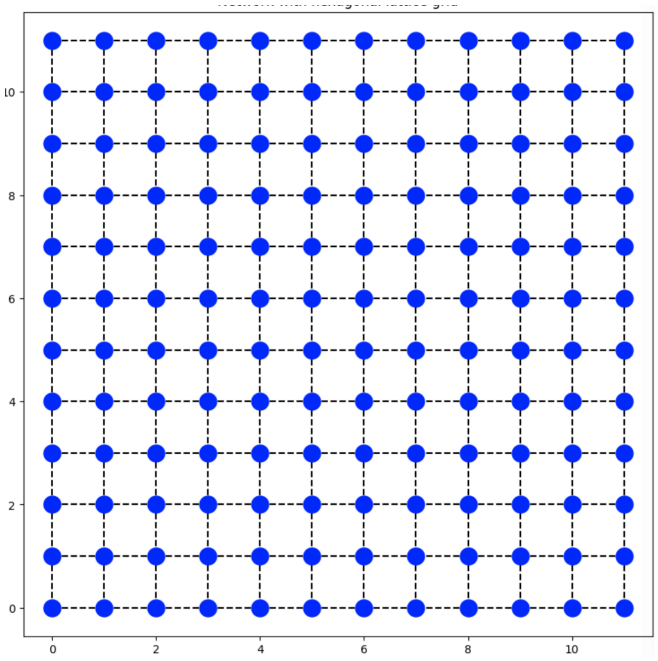
Na koniec zająłem się sprawdzeniem jak na v-measure wpływa szerokość sąsiedztwa. Jest to parametr który reguluje jak bardzo neurony sąsiednie na siatce do neuronu najbliższego są przyciągane. Badania przeprowadziłem na danych cube i pomimo uśrednienia dla każdej wartości parametru po 5 rożnych próbach finalny wykres jest poszarpany. Jak widzimy najlepsze rezultaty daje sąsiedztwo z przedziału [0.4, 0.6], im dalej w lewo tym gorsze rezultaty otrzymujemy, jednak spadek jest delikatny. Inaczej sytuacja przedstawia się po drugiej stronie tego zbioru, podczas gdy na początku do wartości 0.2 spadek również jest stosunkowo delikatny to po przekroczeniu tej wartości gwałtownie spada. Moja teoria jest taka, że oddziaływania pomiędzy sąsiednimi neuronami są wtedy zbyt słabe, aby zadziałała "reguła Hebba" i sieć nie potrafi dopasować swojej topologii do danych wejściowych.



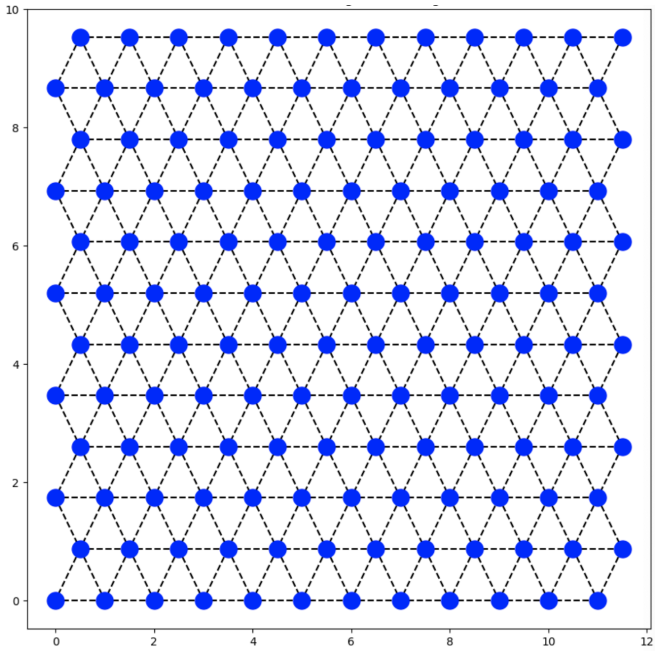
Rysunek 4: Zależność v-measure od szerokości sąsiedztwa

3.2 KOH2

Na ten etap przeznaczony był tylko niecały tydzień, część implementacyjna była równie prosta co pierwsza tylko zdecydowanie krótsza. Polegała ona na możliwości dostosowania sieci do używania siatki hexagonalnej zamiast prostokątnej, w takim przypadku każdy z sześciu sąsiadów od danego neuronu znajduje się w równej odległości. Poniżej załączam wizualizacje obu wariantów struktury warstwy Kohonena. Są one według mnie *visually pleasing*. Pokazują one również jakie współrzędne mają moje neurony "pod maską".



Rysunek 5: Wizualizacja struktury prostokątnej warstwy Kohonena



Rysunek 6: Wizualizacja struktury hexagonalnej warstwy Kohonena

W tym etapie zdecydowałem się też na wykonanie optymalizacji, wiedząc, że będę musiał operować na znacznie większych zbiorach niż cube czy hexagon. Zaimplementowanie jej zajęło mi kilka godzin (tak żeby wszystko działało) jednak dużo się nauczyłem podczas tego procesu i czuję, że lepiej zrozumiałem dzięki temu sieci Kohonena. Mimo, że nie obowiązkowe było to zdecydowanie najtrudniejsze zadanie w tym etapie. O tym czy było warto pisze w dodatku. Model liczy się szybciej ale nie ma tu mowy o przyśpieszeniach, które doświadczałem w Neural Networks. Aby poprawnie zaimplementować wektoryzację musiałem sobie wszystko rozpisać na kartce w postaci macierzowej i zaimplementować krok po kroku, co było nie trywialnym zadaniem.

Naszą implementację musieliśmy w tym etapie przetestować na 2 zbiorach danych:

- MNIST - jednym z najpopularniejszych zbiorów w społeczności Uczenia Maszynowego, jest to zbiór z 70 tysiącami (60k train, 10k test) czarno białych zdjęć o rozmiarze 28 na 28 pikseli przedstawiających odręcznie pisane liczby, przez wielu zbiór ten jest nazywany Hello World'em Machine Learningu,
- Human Activity Recognition Using Smartphones - jest to zbiór mniej popularny niż wyżej wspomniany MNIST, są to dane tabelaryczne zawierające pomiary aktywności ludzi wykonujących codzienne czynności takie jak chodzenie, siedzenie, wchodzenie po schodach, czy leżenie.

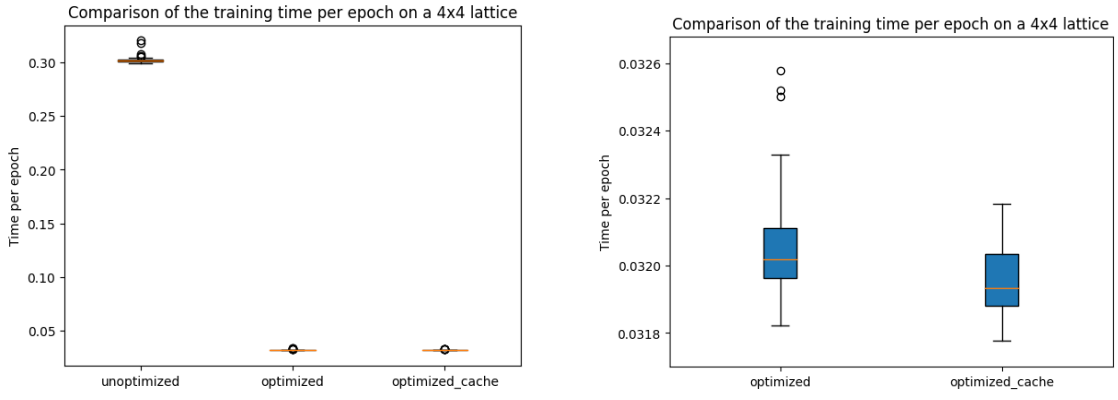
Problem z tymi danymi jest taki, że są wielowymiarowe. Nie możemy więc ich zwizualizować, możemy się jedynie posiłkować metrykami takimi jak v-measure.

3.3 Dodatkowe

W trakcie tych wszystkich zadań wykonałem też kilka eksperymentów które nie pasują tematycznie do poleceń dla danego tygodnia. Postanowiłem, że najciekawsze załączę w tej części. Jest to porównanie szybkości wykonywania obliczeń przed optymalizacją, tak jak zapisany jest kod na slajdach z wykładu, z zagnieżdżonymi pętlami, po optymalizacji, która według mnie nie była trywialna i która pozwoliła na pozbycie się 2 wewnętrznych pętli na rzecz wektoryzacji. Finalną wersją którą będę porównywał jest rozszerzeniem tej z wektoryzacją o cache'owanie macierzy odległości na siatce neuronów, niezależnie o tego czy jest to prostokątna siatka czy heksagonalna. Przeprowadziłem eksperymenty na zbiorach cube i MNIST.

3.3.1 Cube

Na cube przeprowadziłem eksperyment porównujący 3 poziomy optymalizacji dla siatki neuronów 4x4 oraz 9x9, oraz porównujący 2 szybsze wersje dla siatki 25x25, 45x45, 65x65 oraz 90x90.



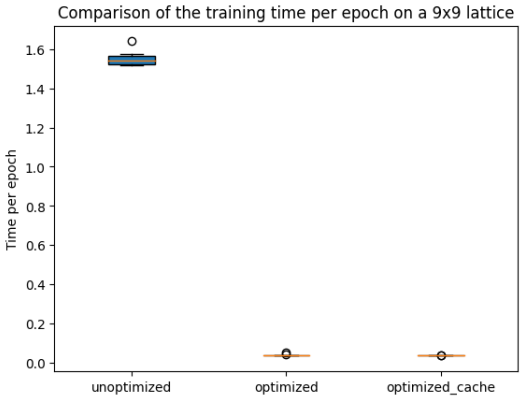
(a) Porównanie 3 metod

(b) Porównanie 2 bardziej optymalnych metod

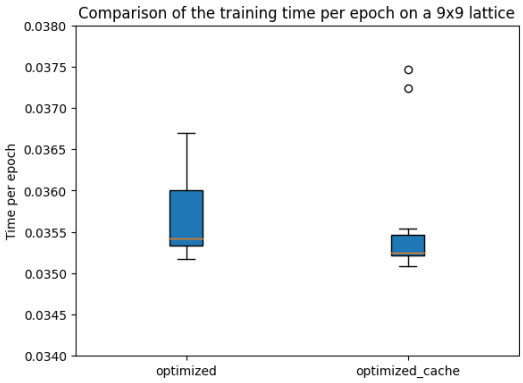
Rysunek 7: Ile sekund liczy się jeden epoch dla siatki 4x4

Musiałem załączyć dwie wizualizacje, żeby w pełni zobrazować otrzymane wyniki. Na powyższych wykresach widać, że ze zoptymalizowane wersje są przy tych parametrach około 10 razy szybsze niż

nie zoptymalizowania wersja, nie tak źle. Niestety dodanie funkcjonalności cache'owania nie poprawiło znacznie wyników. Możliwe, że gdybym zaimplementował to inaczej (wyłącznie w ramach pakietu numpy) to otrzymał bym lepsze rezultaty. Aktualnie jest to zaimplementowane jako słownik w którym kluczami są krotki postaci (x,y), a wartościami macierze sąsiedztwa.



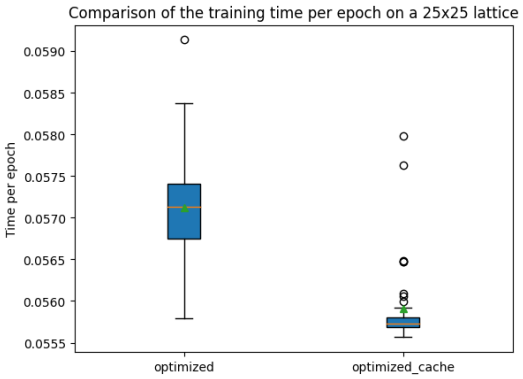
(a) Porównanie 3 metod



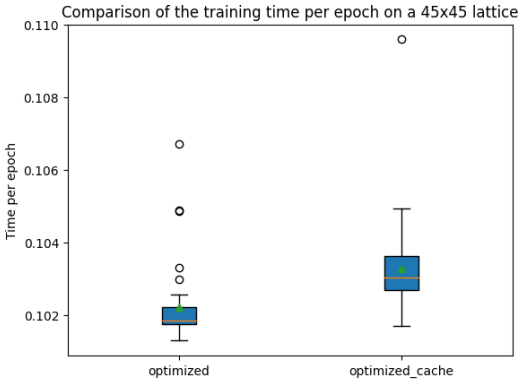
(b) Porównanie 2 bardziej optymalnych metod

Rysunek 8: Ile sekund liczy się jeden epoch dla siatki 9x9

Podobne wyniki uzyskałem dla siatki 9x9 z tym, że teraz optymalizacja przyczyniła się do przyśpieszenia około 50-krotnego. Wersja z cache jest szybsza i wydaje się dawać więcej niż dla przypadku 4x4, jednak nie jest to znacząca różnica. Zacząłem się więc zastanawiać czy wraz ze wzrostem rozmiaru siatki wersja z cache zyskuje coraz więcej względem odpowiednika bez tej funkcjonalności i otrzymałem ciekawe wyniki. Wykresy są stworzone w oparciu o 40 wykonań algorytmu. Pierwotnie nie dowierzałem wynikom jednak odpaliłem notebook jeszcze raz i otrzymałem te same rezultaty.



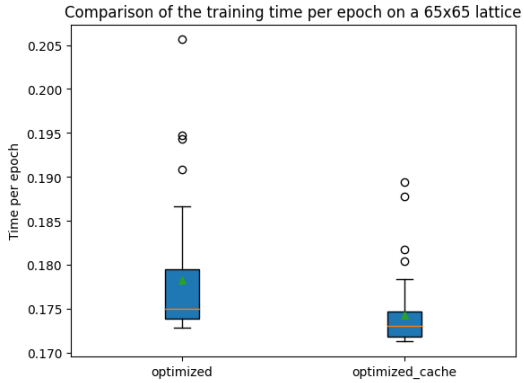
(a) Siatka 25x25



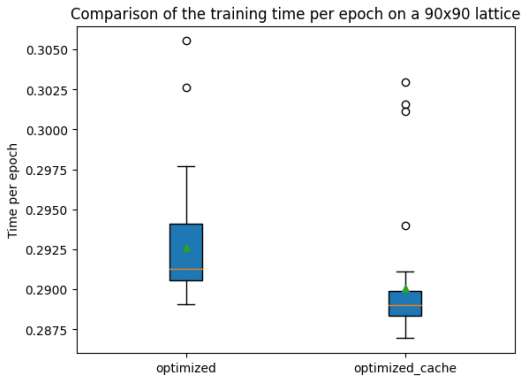
(b) Siatka 45x45

Rysunek 9: Ile sekund liczy się jeden epoch dla różnych rozmiarów siatek

Widzimy, że dla siatki 25x25 cache wyraźnie poprawia czas wykonania natomiast dla siatki 45x45 sytuacja się nieoczekiwanie odwraca i teraz to wersja z cache jest wolniejsza. Wydaje mi się, że te nieoczekiwanie zachowanie spowodowane jest tym jak python przechowuje słowniki, i odczytanie macierzy z pamięci jest wolniejsze niż obliczenie jej sobie od nowa. Sprawdziłem w końcu co się dzieje gdy weźmiemy jeszcze większą siatkę mianowicie 65x65 oraz 90x90. Tutaj najwyraźniej koszt obliczenia macierzy przewyższa wolne odczytanie je z pamięci i wersja z cache znowu jest szybsza. Widać to po tym, że w siatce 90x90 wersja z cache zwiększa nieznacznie swoją przewagę w porównaniu do siatki 65x65.



(a) Siatka 65x65



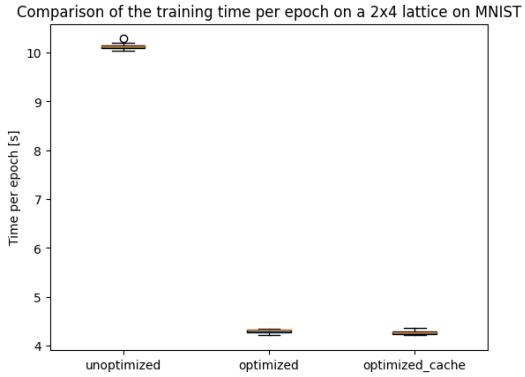
(b) Siatka 90x90

Rysunek 10: Ile sekund liczy się jeden epoch dla różnych rozmiarów siatek

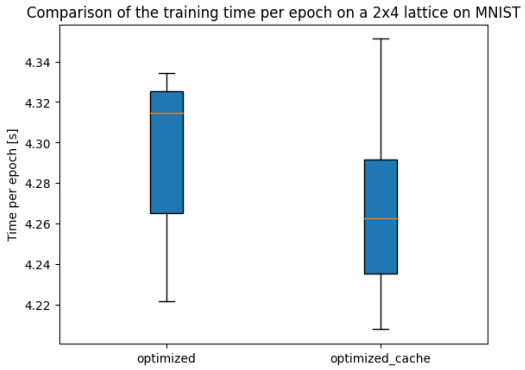
Co warto docenić to jak dobrze skaluje się sieć wraz ze wzrostem liczby neuronów. Sieć nie zoptymalizowania gdy było 81 neuronów liczyła się około 1.5 sekundy. Czas obliczeń rósł w tej implementacji liniowo co do ilości neuronów. Możemy więc oszacować ile zajęła by wersji nie zoptymalizowanej jedna epoka w siatce 90x90. Wykonując obliczenia otrzymujemy wynik 150 sekund, podczas gdy zoptymalizowanym wersjom zajmuje taka operacja 0.3 sekundy. Co ciekawe dla wersji nie zoptymalizowanej przejście z siatki 9x9 na 90x90 skutkuje w 100 zwiększeniu czasu obliczeń natomiast dla wersji zoptymalizowanych jest to tylko 10 krotne zwiększenie.

3.3.2 MNIST

Dla zbioru MNIST sprawdziłem tylko 2 konfiguracje dla wszystkich 3 wersji mianowicie 2x4 i 4x4. Powtarzają się patterny, które już widzieliśmy. Obie wersje zoptymalizowane są do siebie zbliżone podczas gdy nie zoptymalizowana jest wolniejsza, tym razem tylko 2.5 raza. Jest to spowodowane małym rozmiarem siatki.



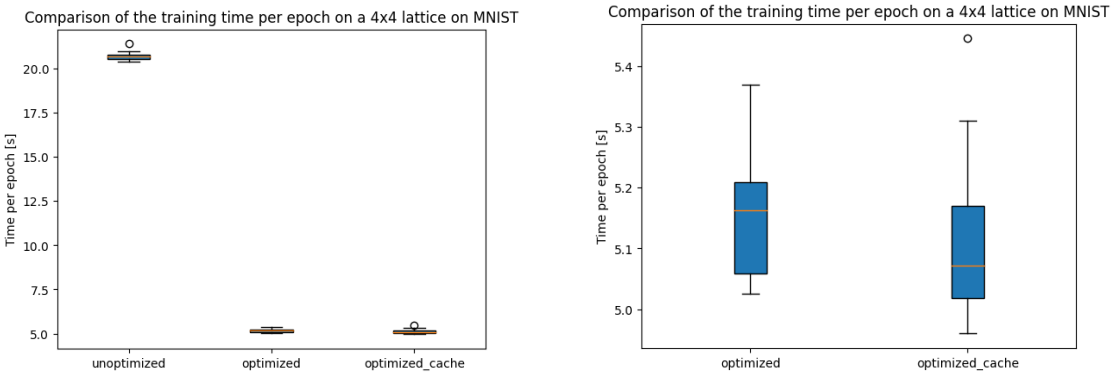
(a) Porównanie 3 metod



(b) Porównanie 2 bardziej optymalnych metod

Rysunek 11: Ile sekund liczy się jeden epoch dla siatki 2x4

Zwiększenie siatki dwukrotnie powoduje dwukrotne zwiększenie czasu obliczeń nie zoptymalizowanej wersji (2 razy dłuższa pętla) tak więc teraz zoptymalizowane wersje są około 4 razy szybsze.



(a) Porównanie 3 metod

(b) Porównanie 2 bardziej optymalnych metod

Rysunek 12: Ile sekund liczy się jeden epoch dla siatki 4x4

4 Podsumowanie

Ten projekt również oceniam za udany. Mówiąc prawdę był on bardzo czasochłonny, jednak czas ten nie poszedł na marne. Nauczyłem się niezwykle dużo z obszaru o którym nie miałem za dużo pojęcia. Zadania uważam za wykonane poprawne, otrzymywałem zadowalające wyniki. Jestem zadowolony z tego co zrobiłem, a robiąc to, dobrze się bawiłem, szczególnie podczas wykonywania optymalizacji. Jednak będąc szczery wolałem temat sieci neuronowych niż Kohonena, może to kwestia tego, że ten etap trwał tylko 3 tygodnie, może dlatego, że jest to bardziej niszowy temat i nie ma w internecie tylu materiałów pozwalających dogłębnie zrozumieć dane zagadnienie.