Dokumentacja końcowa

Podstawy Sztucznej Inteligencji

Autorzy
Antoni Charchuła
Michał Witkowski
Jakub Tokarzewski

Spis treści

1. Wstęp

Naszym poleceniem projektowym jest „Stworzyć, wytrenować i przeprowadzić walidację sieci neuronowej, która dokona predykcji, czy dane wino jest dobrej jakości: https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009 ”.
Jest to przykład problemu klasyfikacji za pomocą perceptronu wielowarstwowego, czyli grupowania win na podstawie jakości ich wybranych cech. Każde wino przedstawione jest za pomocą 12 cech, z których każde jest ocenione w innej skali. Finalna jakość wina określona jest przez ocenę w skali od 0 do 10. Wina podzieliliśmy na dwie kategorię, dobre (ocena większa niż 6.5) oraz słabe (ocena mniejsza niż 6.5).

1.1. Technologia

Aplikacja zostanie zrealizowana w języku Python z użyciem biblioteki *numpy,* wspomagającej prowadzenie obliczeń na macierzach. Podjęliśmy próbę przeprowadzania obliczeń na karcie graficznej Nvidii za pomocą biblioteki Cupy. Niestety ta funkcjonalność nie działa, ponieważ z niewiadomych dla nas przyczyn, wszystkie obliczenia i tak są prowadzone na procesorze.

1.2. Repozytorium

Wszystkie pliki źródłowe można na znaleźć w naszym repozytorium pod adresem https://github.com/tokerson/PSZT-Wine_Classification.

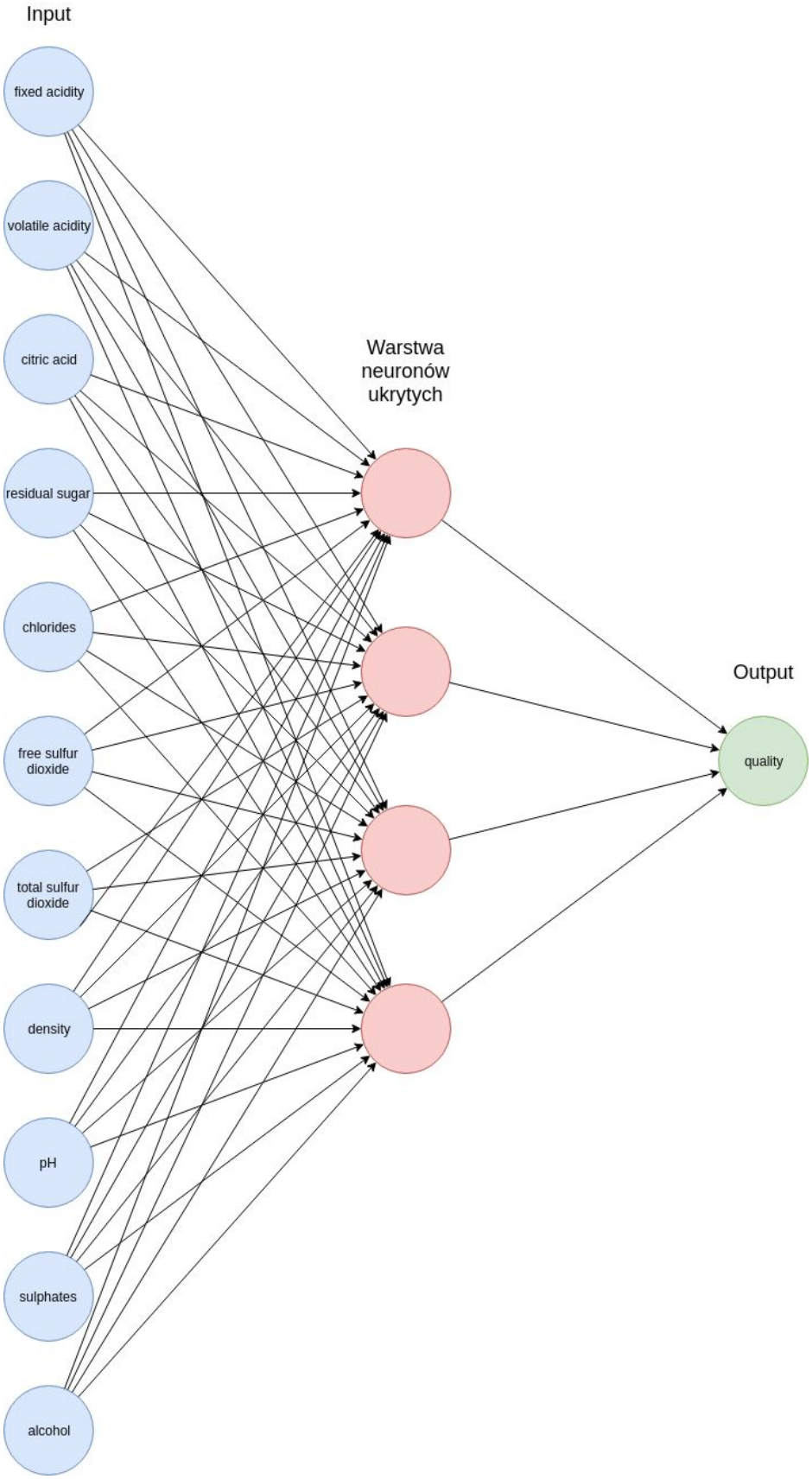
2. Etapy projektu

Projekt podzieliliśmy na kilka mniejszych etapów:

1. Stworzenie modelu perceptronu dwuwarstwowego. ~ JT
2. Implementacja obliczania wyjścia sieci neuronowej na podstawie wejść. ~ AC
3. Wczytywanie zestawu danych z pliku. ~ JT , AC
4. Implementacja propagacji wstecznej za pomocą metody stochastycznego najszybszego spadku. ~ AC
5. Podział danych na zbiory treningowe i testowe ~ JT
7. Badanie sieci neuronowej pod względem przeuczenia oraz zbyt małego dopasowania. ~ JT, AC, MW
8. Tworzenie wykresów obrazujących celność sieci ~ JT
9. Stworzenie trybu przeprowadzającego obliczenia przy pomocy Cupy ~ MW
10. Stworzenie interfejsu aplikacji ~ MW

AC - Antoni Charchuła, MW - Michał Witkowski, JT - Jakub Tokarzewski

3. Realizacja zadania



3.1. Inicjalizacja sieci

Po analizie budowy i działania perceptronu zdecydowaliśmy się, że warstwa ukryta naszej sieci będzie początkowo miała wielkość $\sqrt{|X|}$ (gdzie X to zbiór wejściowy). Liczba neuronów w warstwie ukrytej może ulec zmianie w związku z testami przeprowadzanymi w pkt. 7 planu realizacji projektu.

Początkowe wagi neuronów warstwy ukrytej to liczby z rozkładu

$$U \sim \left(\frac{-1}{\sqrt{(dimX)}}, \frac{1}{\sqrt{(dimX)}}\right)$$

gdzie dim(X) to wymiar wejścia sieci.

Początkowe bias-y są zerowe.
Początkowe wagi krawędzi prowadzących z warstwy ukrytej do neuronu wyjściowego z rozkładu $U \sim \left(\frac{-1}{\sqrt{(dimH)}}, \frac{1}{\sqrt{(dimH)}}\right)$, gdzie dim(H) to liczba neuronów w warstwie ukrytej.

3.2. Funkcja aktywacji

Funkcja aktywacji użyta w naszej sieci to *sigmoid* wyrażona wzorem:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Ta sama funkcja aktywacji stosowana w warstwie ukrytej oraz na wyjściu sieci. Dzięki temu wynik będzie znormalizowany do przedziału 0 – 1.

3.3. Propagacja wsteczna

Do nauki sieci zdecydowaliśmy się skorzystać z metody stochastycznego najszybszego spadku, ze względu na to, że jest szybsza niż metoda gradientu prostego, którą również rozważaliśmy. Ostateczna decyzja spowodowana była tym, iż mimo że w teorii metoda stochastycznego najszybszego spadku daje gorsze przybliżenie od metody gradientu prostego, to nasza sieć przy jej użyciu osiągnęła zadowalającą celność w okolicy 85%. Formuła uczenia się ma w tej metodzie postać:

$$\theta_{t+1} := \theta_t - \beta_t \frac{d}{d\theta_t^T} \frac{1}{2} \left\| \overline{f}(x_t; \theta_t) - y_t \right\|^2$$

Aktualizacja wag i bias-ów odbywa się korzystając z powyższej zależności, gdzie β to współczynnik tempa uczenia, który został eksperymentalnie wyznaczony na etapie testowania sieci neuronowej. Zbyt duży współczynnik będzie prowadził do zbyt dużych skoków i w efekcie do niedokładnego przybliżania jakości wina, a zbyt mały do zbyt wolnego tempa nauki sieci. Uczenie będzie następowało w kilku epokach, których ilość również została dobrana. Wartości wszystkich dobranych współczynników zostały opisane w punkcie 3.6.

3.4. Interpretacja wyników

W warstwie wyjściowej naszego perceptronu mamy jeden neuron mogący przyjąć wartość od 0 do 1. Oceny wina są w skali 0 – 10, więc po przeskalowaniu ocen do zakresu 0 – 1, będziemy mogli stwierdzić jakiej dane wino jest jakości porównując je z wyjściem. Wina z oceną poniżej 0.65 zostają klasyfikowane jako słabe, a te z oceną powyżej 0.65 jako dobre. Podjęliśmy również próby podziału win na trzy klasy (słabe - ocena < 4.5, średnie - ocena (4.5; 6.5), dobre ocena > 6.5). Celność naszej sieci w takim wypadku wahała się w okolicach 68%, co nie było dla nas zadowalającym wynikiem, więc odeszliśmy od tego pomysłu.

3.5. Normalizacja wejść

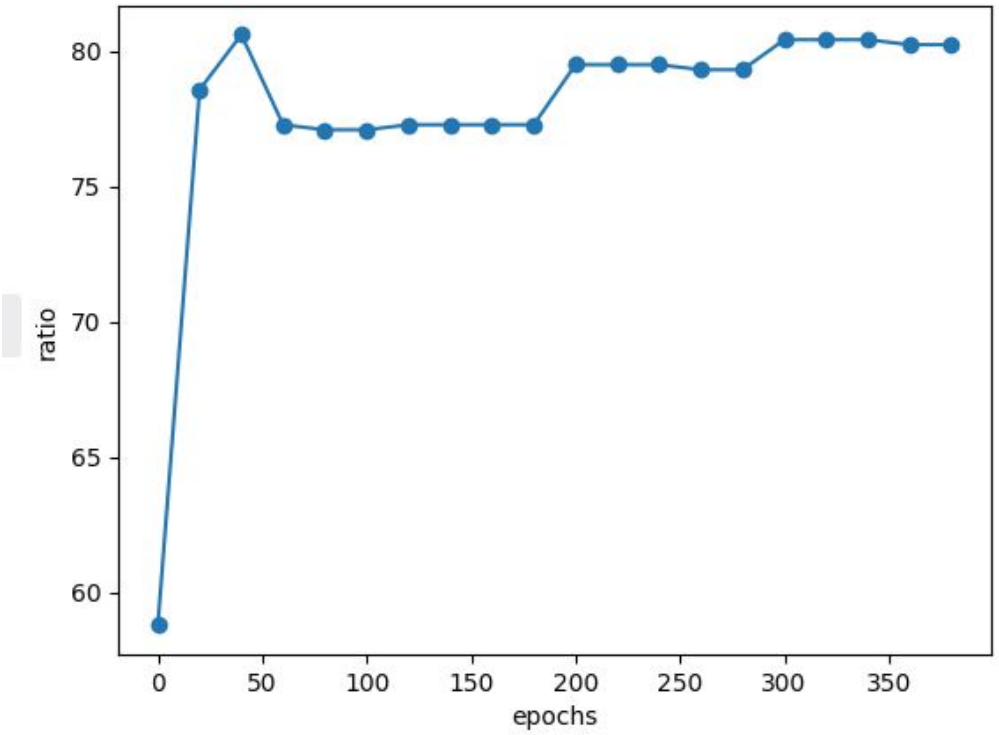
Dane wejściowe są podane w postaci 11 parametrów, które są ocenami danej cechy wina oraz 12-tego parametru, będącego oceną jakości wina. Każda z tych ocen jest z innego przedziału, więc mają różny wpływ na wynik obliczany przez sieć. Aby temu zapobiec wejścia zostały przeskalowane do przedziału 0 – 1.

3.6. Trening i testowanie

Kluczowym elementem projektu jest testowanie zaimplementowanej sieci neuronowej i najważniejszych dla jej działania wartości (β – współczynnik tempa uczenia, liczba epok). Na tym etapie nastąpiło dobranie odpowiednich współczynników na podstawie badania dokładności klasyfikowania win przez sieć neuronową oraz badanie wyników pod względem przetrenowania.

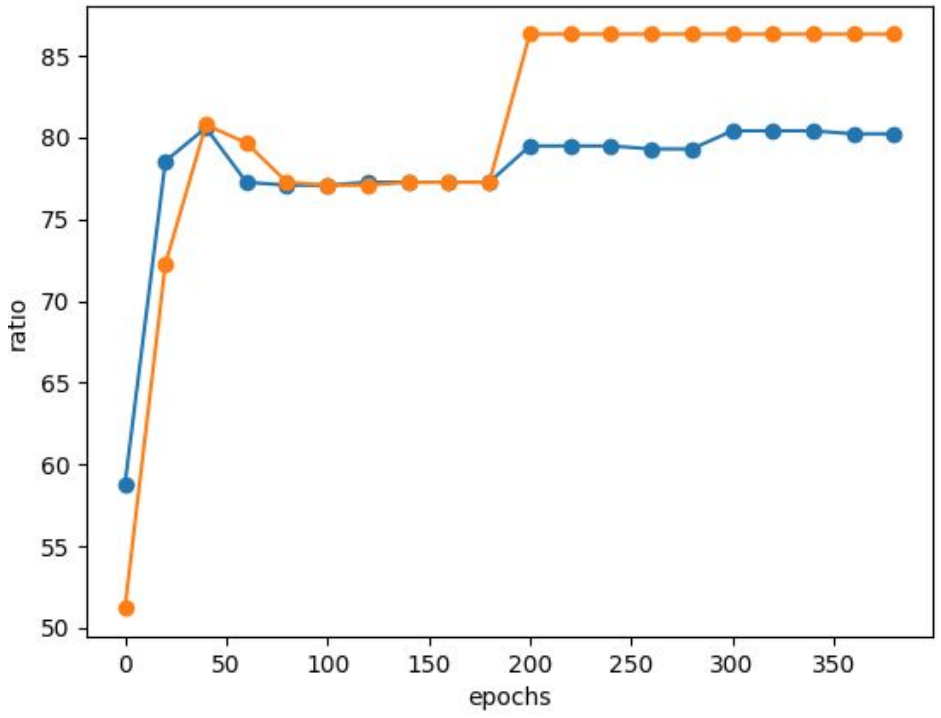
Podczas testowania działalności sieci podzieliliśmy zbiór wszystkich win na dwa podzbiory, wina dobre (217) i wina złe (1382). Zbiory te nie są równoliczne, więc aby wyniki pomiarów były miarodajne, to “klonowaliśmy” wina dobre, aby było ich tyle samo co win złych. Zbiór treningowy stanowi 80% win dobrych i 80% win złych, a reszta win to zbiór testowy.

Przed trenowaniem wymieszaliśmy zbiór treningowy.
Trenowanie zaczęliśmy od współczynnika $\beta = 0.1$ i zmniejszaliśmy go dla kolejnych testów.



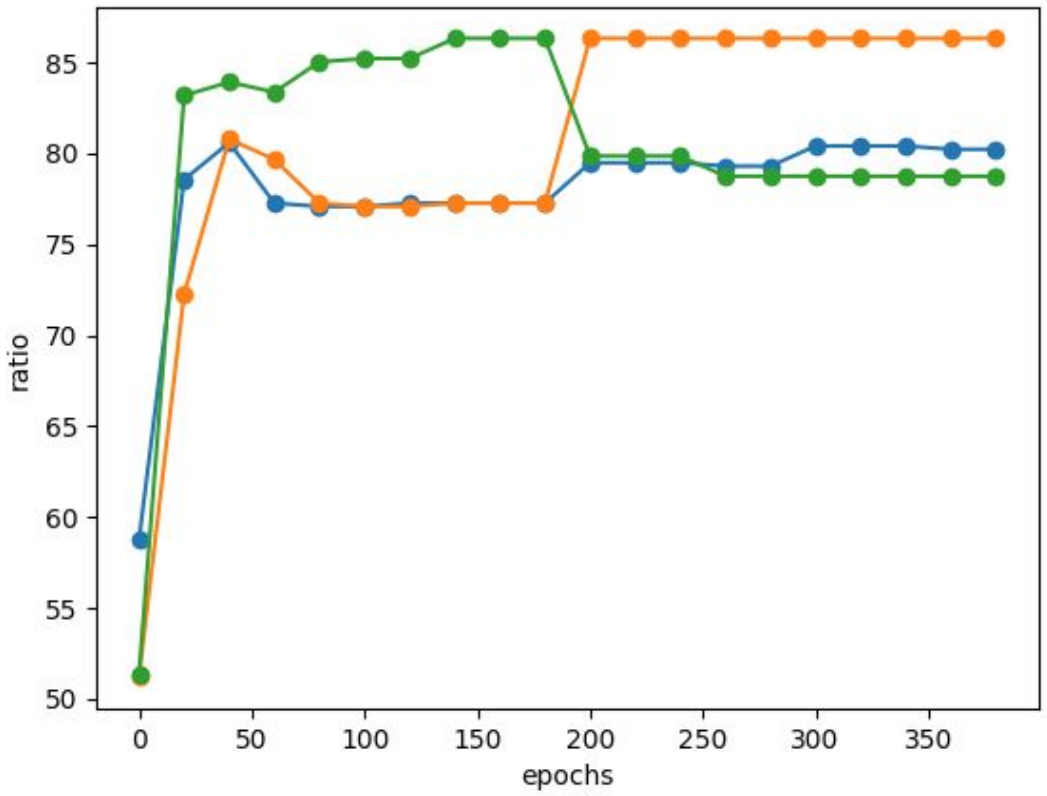
Learning rate = 0.1

Jak widać na wykresie powyżej sieć nie osiągnęła celności większej niż 80% i sieć szybko przestała się uczyć.



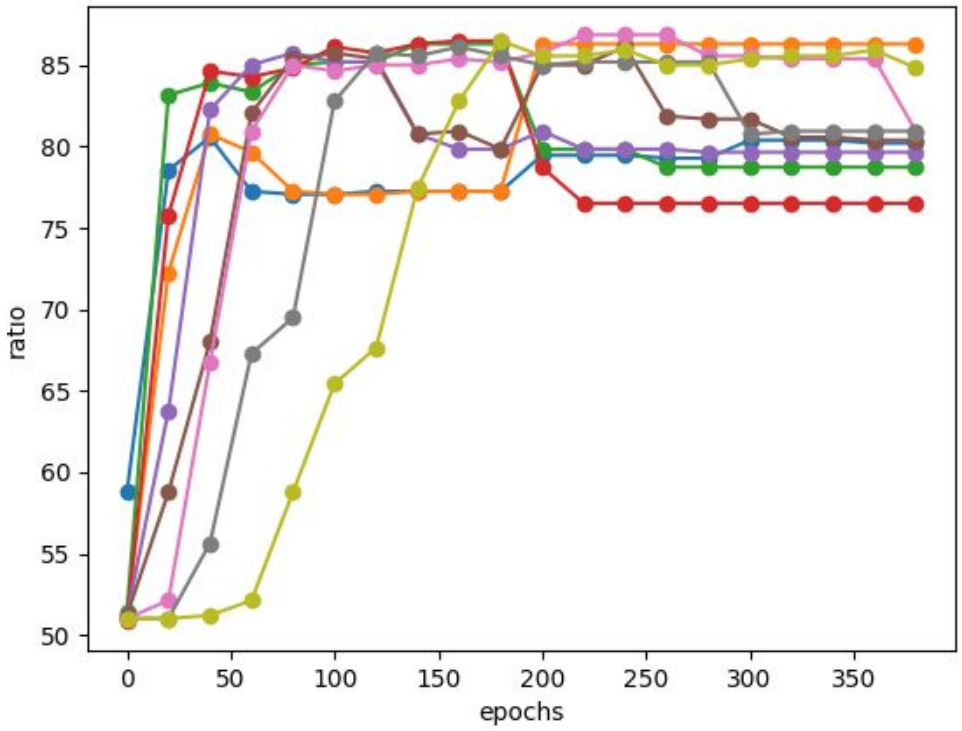
Learning rate = 0.09 Linia pomarańczowa

Po zmniejszeniu *Learning rate* do 0.09 widzimy poprawę celności sieci do 85%, jednak uczenie następuje powoli, co może świadczyć o zbyt dużym współczynniku (bo zmiany są zbyt duże i nie poprawiają celności), lub o zbyt małym współczynniku. Na kolejnym wykresie zostanie pokazane działanie sieci z *Learning rate* na poziomie 0.08.



Learning rate = 0.08 Linia zielona

Powyższy wykres napawa optymizmem, gdyż widać znaczącą poprawę tempa uczenia i celność 85% osiągamy już po ok. 100 epokach i jest to dla tego współczynnika maksymalna osiągnięta celność. Po 200 epoce zauważalny jest znaczny spadek celności, co może świadczyć o przeuczeniu sieci. Poniżej został przedstawiony również wykres obrazujący 10 różnych *Learning rate.*



Linia czerwona - lr = 0.07, Linia fioletowa - lr = 0.06, Linia brązowa - lr = 0.05, Linia różowa - lr = 0.04, Linia szara - lr = 0.03, Linia żółta - lr = 0.02

Z powyższego wykresu wyraźnie widać, że mimo zmniejszania *Learning rate* do poziomu 0.02 nie osiągnęliśmy poprawy celności sieci, której maksymalna wartość plasuje się w okolicach 85%. Dobrze widoczny jest również wpływ zmniejszenia *Learning rate* na tempo uczenia się sieci. Dla kolejnych wartości celność na poziomie 85% osiągana jest coraz wolniej, dla *Learning rate = 0.02 (*linia żółta) dzieje się to dopiero po około 175 epoce.

Korzystając z powyższych wykresów zdecydowaliśmy, że najbardziej optymalnie pod względem tempa uczenia i celności sieci będzie wybranie współczynnika *Learning rate* równego 0.08 i zakończenie nauki po 100 epokach.

3.7. Implementacja

W pliku *Network.py* znajduje się implementacja sieci neuronowej. Można tam znaleźć także funkcje pozwalające na trening sieci.

- *def sigmoid* - funkcja licząca sigmoid z podanej liczby
- *def sigmoid_derivative* - funkcja licząca pochodną funkcji sigmoid
- *def feed_forward* - funkcja, jako parametr pobiera zbiór parametrów wina i zwraca wynik od 0-1, mówiący o jakości wina. Jakość wina obliczana jest za pomocą wzoru:

$$\hat{y} = \sigma(W_2 \sigma(W_1 x + b_1) + b_2)$$

- *def backward_propagation* - funkcja realizująca propagację wsteczną w sieci. Jako parametr przyjmuje oczekiwaną jakość, zwróconą jakość przez funkcję *feed_forward* oraz parametry wina.
  Na samym początku obliczana jest wartość funkcji kosztu (*output_loss*). W następnych etapach aktualizujemy wagi W1 i W2.
  Na ostateczną wartość funkcji kosztu mają wpływ wagi poszczególnych neuronów - im większa waga tym większy wpływ miał dany neuron. Dlatego na samym początku musimy określić jaka część finalnego błędu należy do poszczególnego neuronu ukrytego. Dane te zapisujemy w zmiennej *hidden_loss*, którą inicjializujemy następująco:

  Dla każdego neuronu ukrytego dodajemy do *hidden_loss* wartość:
  *output_loss * waga_neuronu * sigmoid_derivative(wartość neuronu ukrytego)*

  Wartości neuronów ukrytych zapiujemy w trakcie *feed_forward* w zmiennej A1.

  Aktualizacja wag W1 (czyli krawędzi łączących input z neuronami ukrytymi) polega na dodaniu do W1 macierzy tej samej wielkości zainicjalizowanej:

  ```
  for i in range(0, input_size):
    for j in range(0, hidden_size):
      update_W1[i][j] = hidden_loss[j] * input[i] * learing_rate
  ```

  Analogiczna operacja będzie przebiegała pomiędzy wartwą ukrytą, a wartwa wyjściową, tylko jako, że w wartswie wyjściowej występuje tylko jeden neuron, to obliczenia będą prostsze. Wystarczy dodać do W2, tablice o tym samym wymiarze zainicjalizowanej:

  ```
  for i in range(0, self.hidden_size):
    update_W2[i][0] = output_loss * wartość danego neuronu ukrytego * learning_rate
  ```

  Aktualizacja biasów pomiędzy warstwą input, a warstwą neuronów ukrytych polega na dodaniu do B1 macierzy o takim samym wymiarze, która zawiera w sobie tablicę hidden_loss z przemnożonymi wartościami przez learning_rate. Analogicznie do B2 dodajemy wartość output_loss przemnożoną przez learning_rate.

Plik *NetworkCupy.py* zawiera implementację wyżej opisanej sieci neuronowej przy użyciu bilbioteki Cupy. Niestety obliczenia przy użyciu tej implementacji prowadzone są z niewiadomyh dla nas przyczyn, bardzo wolno. Pomimo tego, że obliczenia powinny być prowadzone na GPU, w menadżerze zadań widać, że wydajność GPU cały czas jest na poziomie 0 %.

3.8. Wnioski

W przypadku biblioteki Cupy prawdopodobnie źle skonfigurowaliśmy środowisko, dlatego nam to nie działało. Możliwe też, że nieodpowiednio zaimplementowaliśmy funkcje przeprowadzające obliczenia na karcie graficznej.

W przypadku normalnej sieci, wynik na poziomie 85% jest zadowalający. Jednak istnieją metody, które dałyby na pewno znacznie lepszy wynik. Wybraliśmy metodę stochastycznego najszybszego spadku, ze względu na jej prędkość obliczeń i satysfakcjonującą celność klasyfikacji.

W punkcie 3.6. zostały przedstawione przemyślenia na temat doboru współczynników w sieci i ich wpływu na celność oceniania win. Finalnie dobrane wartości to:

Learning rate = 0.08
Liczba epok = 100

Aktualnie w przypadku treningu zwracana jest ostatnia postać sieci po wykonanych epokach, nieważne czy straciła ona na jakości przez zjawisko przeuczenia. Można zmienić implementację na taką, która zapamiętuje w trakcie treningu sieć, która osiąga najlepsze wyniki i zwraca właśnie tą.