Michał Przysucha

Kacper Radzikowski

WMH – projekt

PB2. Zastosowanie sieci neuronowej w zadaniu aproksymacyjnym

Wstępne ustalenia:

1. Język programowania: Python

2. Dane uczące i testujące: wygenerowane z nietrywialnej funkcji

3. Etapy:

- implementacja sieci w różnych strukturach (liczba warstw ukrytych oraz liczba neuronów w warstwie)

- uczenie sieci, w tym zbadanie procesu uczenia się sieci

- określenie optymalnej struktury

1.1 Oznaczenia użyte w opisie oraz w kodzie źródłowym

- liczba wartsw w sieci neuronowej

- indeks warstwy w sieci neuronowej

- liczba węzłów w l-tej warstwie sieci neuronowej, w szczególności:

- liczba węzłów w 0-wej warstwie (warstwie wejściowej) sieci neuronowej (liczba cech danych trenujących – w naszym zadaniu )

- liczba danych trenujących

- indeks danych trenujących (np. i-ty przykład w zbiorze trenującym)

- wartość na wyjściu l-tej warstwy (po funkcji aktywacji) dla i-tej danej trenującej. W rzeczywistości implementacja będzie opierać się na macierzach, zatem skorzystamy z notacji jak niżej:

- macierz wyjść l-tej warstwy dla wszystkich danych trenujących

- wartość na wejściu sieci neuronowej dla i-tej danej trenującej  
- wartość na wejściu sieci neuronowej (macierz wejść wszystkich danych trenujących lub testowych – w wierszach będą cechy zmiennej objaśnianej (w tym zadaniu będą 2 wiersze), a w kolumnach kolejne dane trenujące lub testowe)

- wartość na wyjściu wielomianiu w l-tej warstwie sieci neuronowej (przed funkcją aktywacji) dla i-tej danej trenującej. Tu również implementacja będzie używać macierzy, zatem skorzystamy z poniższej notacji:

- macierz wyjść wielomianiu w l-tej warstwie sieci neuronowej (przed funkcją aktywacji) dla wszystkich danych trenujących

- funkcja aktywacji w l-tej warstwie ()

- odpowiednio: wagi i wyraz wolny (bias) w l-tej warstwie sieci neuronowej (współczynniki wielomianu wykorzystane do policzenia , konkretnie: )

- wyjście L-tej warstwy (wyjściowej) sieci neuronowej dla i-tej danej trenującej (jest to predykcja dla danych wartości współczynników podczas procesu uczenia, która posłuży do policzenia kosztu, lub predykcja dla finalnych (wytrenowanych) współczynników dla danych testowych, ponieważ korzystamy z obliczeń na macierzach, w rzeczywistości będzie uzywać poniższej zmiennej:

- wartość na wyjściu L-tej warstwy (wyjściowej) sieci neuronowej (predykcja), jest to macierz wyjść wszystkich danych trenujących.

- gradienty dla l-tej warstwy sieci neuronowej

1.2 Wymiary macierzy i wektorów

W poprzednim punkcie przedstawione zostały podstawowe zmienne, które zostaną użyte podczas implementacji. Ponieważ w wielu miejscach zostało zaznaczone, że użyte będą te wersje zmiennych, które są w postaci wektorów i macierzy, poniżej zaprezentowane zostały wymiary wspomnianych wektorów i macierzy.

Wykorzystujemy wektory i macierze z biblioteki NumPy. Potrafią one wykorzystać instrukcje Streaming SIMD Extentions (Single Instruction Multiple Data) procesora, dzięki czemu obliczenia wykonują się dużo szybciej, niż gdybyśmy obliczali je iteracyjnie dla każdej pojedynczej danej (czyli wykorzystują pętle: for i in range(0,m)).

, - liczba wierszy jest taka sama jak liczba węzłów w l-tej warstwie (wymiar wektora wyjściowego z l-tej warstwy dla pojedynczej danej trenującej), liczba kolumn jest taka sama jak liczba danych trenujących.

Implementacja: obie zmienne będą zatem macierzami o wymiarach:.

- macierz współczynników, liczba wierszy jest taka sama jak liczba węzłów w l-tej warstwie (wymiar wektora wyjściowego z l-tej warstwy dla pojedynczej danej trenującej), liczba kolumn jest taka sama jak liczba węzłów w (l-1)-tej warstwie (wymiar wektora wyjściowego z (l-1)-tej warstwy / wejściowego do l-tej warstwy).

Implementacja: będzie słownikiem / tablicą asocjacyjną, gdzie kluczem będzie (numer warstwy sieci neuronowej), a wartością macierz .

- wektor wyrazów wolnych, liczba wierszy jest taka sama jak liczba węzłów w l-tej warstwie (wymiar wektora wyjściowego z l-tej warstwy)

Implementacja: będzie słownikiem / tablicą asocjacyjną, gdzie kluczem będzie (numer warstwy sieci neuronowej), a wartością wektor .

- w niniejszym zadaniu pojedynczy przykład danej trenującej jest wektorem dwuwymiarowym (dwuelementowym), natomiast będzie macierzą, gdzie wiersze to cechy zmiennej wejściowej, a kolumny to kolejne przykłady danych trenujących, czyli będzie to macierz o wymiarach: .

- macierz o 1 wierszu (dla pojedynczej danej wejściowej, wyjście sieci neuronowej w naszym zadaniu to liczba rzeczywista) oraz o kolumnach, czyli: .

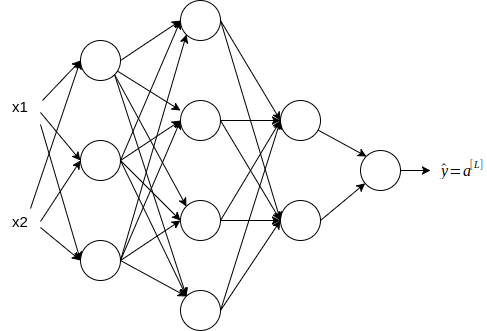
- wyjście wielomianu w l-tej warstwie (przed funkcją aktywacji) dla wszystkich danych trenujących, liczba wierszy jest taka sama jak liczba węzłów w l-tej warstwie (wymiar wektora wyjściowego z l-tej warstwy dla pojedynczej danej trenującej lub testującej), a liczba kolumn jest taka sama jak liczba danych trenujących lub testujących: .

- wyjście funkcji aktywacji w l-tej warstwie, liczba wierszy jest taka sama jak liczba węzłów w l-tej warstwie, a liczba kolumn odpowiada liczbie danych trenujących lub testujących: .

- wymiary gradientów, są takie same jak wymiary odpowiadających im macierzy / wektorów: , czyli np. wymiar macierzy jest dokładnie taki sam jak macierzy itd.

1.3. Przykład

Poniżej zaprezentowany został przykład sieci neuronowej o architekturze jak na poniższym schemacie:



W powyższej sieci neuronowej mamy 3 warstwy ukryte oraz warstę wyjściową, zatem L = 4 (warstwy wejściowej nie liczymy). W poszczególnych warstwach mamy liczbę węzłów:

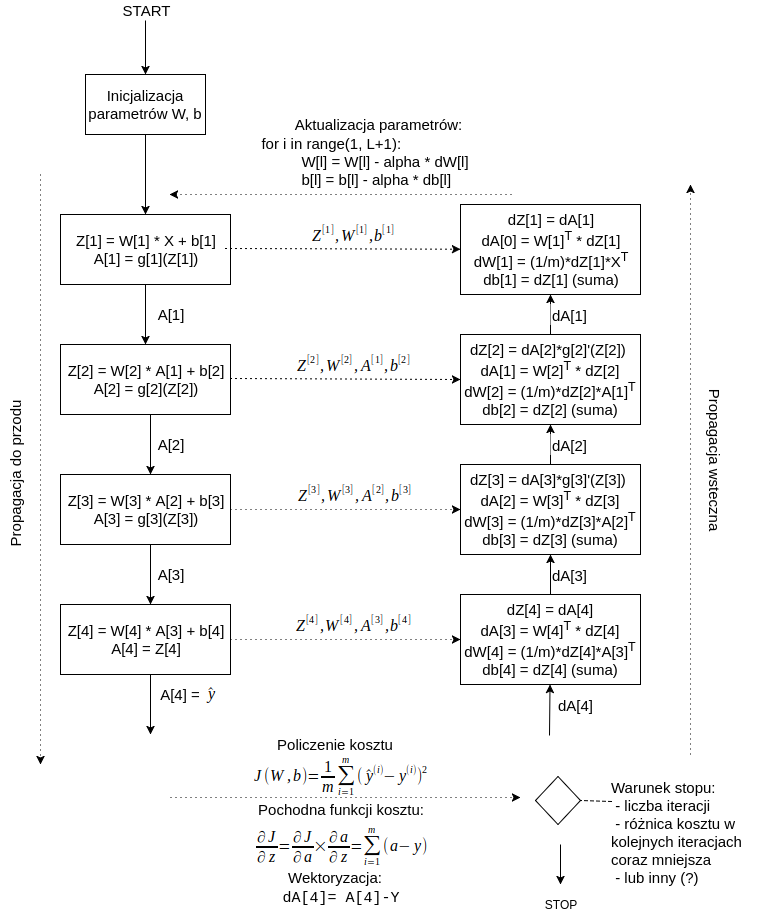
Zmienna wejściowa jest dwuwymiarowa (dwuelementowy wektor), zatem .

Wymiary macierzy ze współczynnikami:

Wymiary wektorów z wyrazami wolnymi:

1.4. Opis algorytmu

Implementacja – przykład dla sieci 4-warstwowej (warstwy zaprezentowane zostały horyzontalnie).



Opis kolejnych kroków:

Zanim zaczniemy trenowanie sieci neuronowej należy przygotować zbiór danych uczących, a także zbiór danych testujących, które posłuży ocenie jakości nauczonej sieci neuronowej. Ponieważ w zadaniu estymujemy pewna funkcję, danej takie możemy wygenerować sami w dużej ilości.

Dodatkowo należy ustalić wartości hiperparametrów: ile warstw sieci, ile węzłów na każdej warstwie, jak wielkości współczynnika szybkości uczenia – . Dwa pierwsze nazwijmy architekturą sieci neuronowej. Będziemy uczyć sieci w różnych architekturach (różna liczba warstw i węzłów na każdej warstwie). Dla każdej architektury będzie próbować różnych wartości współczynnika uczenia . Będziemy monitorować proces uczenia (więcej w nastepnym punkcie 1.5) i na bieżąco decydować, czy przerwać proces z powodu zbyt małej / dużej wartości współczynnika .

Poniższe kroki opisują proces uczenia na zbiorze trenującym:

1. Algorytm zaczyna się od inizjalizacji parametrów (przynajmniej dla muszą to być wartości losowe).

2. Następnie wykorzystując współczynniki przechodzimy przez kolejne warstwy sieci neuronowej, aż obliczymy , czyli estymację (predykcję) . Na każdej warstwie oprócz wyjściowej będziemy używać funkcji aktywacji. Spróbujemy różnych i zobaczymy jakie będą rezultaty. Spróbujemy m.in.: ReLU i Sigmoid.

Na warstwie wyjściowej funkcji aktywacji nie będzie, dlatego, że chcemy, aby wyjście było liczbą rzeczywistą, więc potraktujemy wynik wielomianu (Z[L] = W[L] \* A[L] + b[L]) jako wyjście ostatniej warstwy (A[L] = Z[L]).

3. Nastepnie policzony zostanie koszt. Funkcja kosztu jest tutaj taka sama jak dla regresji, ponieważ nie wykorzystujemy funkcji aktywacji (w szczególności nie wykorzystujemy funkcji sigmoidalnej) na wyjściu. Wkolejnym kroku korzystając z pochodnej funkcji kosztu obliczamy dA[L], które posłuży jako wejście dla propagacj wstecznej.

4. Tutaj pojawi się warunek STOP-u. Może to być z góry ustalona liczba iteracji. Sensowniejsze jednak wydaje się sprawdzanie jak bardzo w kolejnych iteracjach spada policzony w poprzednim kroku koszt. Jeśli koszt przestaje spadać lub spadek zaczyna być niezauważalny lub co gorsza koszt zaczyna rosnąć jest to przesłanka to tego, aby zakończyć proces uczenia. Jeśli nie kończymy procesu uczenia, to przechodzimy do kolejnego kroku, czyli propagacji wstecznej.

5. Podczas propagacji wstecznej liczymy gradienty. Będziemy używać wartości policzonych podczas liczenia estymacji (predykcji) w kroku 2, które dla każdej warstwy zostaną zapisane na boku. Na wyjściu propagacji wstecznej dostaniemy gradienty oraz dla każdej z warstw.

6. Używamy gradientów oraz do aktualizacji parametrów z wykorzystaniem współczynnika szybkości uczenia ( - w kodzie źródłowym będzie to zmienna alpha).

Po aktualizcji parametrów ponownie wykorzystamy sieć neuronową i zaktualizowane współczynniki to policzenia estymacji (predykcji), czyli do policzenia . Innymi słowy wracamy do kroku 1.

Po zakończeniu uczenia otrzymujemy dla każdej warstwy sieci macierz współczynników W[L] oraz wektor b[l] (wyraz wolny). Możemy użyć zbioru testowego, aby ocenić jakość sieci neuronowej. W celu oceny możemy użyć wskaźników oceniających regresję (np. błąd średniokwadratowy MSE lub współczynnik determinacji R2).

1.5. Badanie procesu uczenia sieci neuronowej.

W celu zbadania procesu uczenia sieci będziemy co ustaloną liczbę iteracji (np. 100 albo 500) wyświetlać kosz, dzięki czemu będziemy mogli widzieć, czy koszt rzeczywiście maleje i o ile maleje w poszczególnych krokach.

Dodatkowo po zakończeniu procesu uczenia sieci neuronowej możemy wyświetlić wykres obrazujący koszt w zależności od numeru iteracji (idealnie by było gdyby wyszła krzywa spadająca asymptotycznie do jakieś niedużej wartości).

1.6 Implementacja

Używamy języka Python oraz biblioteki NumPy do obliczeń na wektorach i macierzach. Dodatkowo możmy użyć innych bibliotek – np. w celu wyświetlenia wykresu kosztu w zależności od iteracji.

Implementacja nastąpi zgodnie z powyższym opisem w szczególności diagramem, na którym widoczne są wzory na poszczególnych warstwach. Oczywiście pojedyncza warstwa zostanie zaimplementowana generycznie i reużyta tyle razy ile warstw będzie miała badana architektura sieci neuronowej.

Pliki w projekcie (może się to jeszcze zmienić podczas implementacji):

nn.py – główny plik spinający w całość poszczególne mniejsze kroki z plików opisanych poniżej. W tym pliku nastąpi m.in. inicjalizacja parametrów, pojawi się główna pętla aplikacji, gdzie pojedyczna iteracja będzie oznaczała przejście przez policzenie estymacji (predykcji), następnie kosztu, propagacji wstecznej oraz aktualizacji parametrów. Pojawi się tutaj również warunek STOP‑u. „Klocki” algorytmu takie jak policzenie estymacji (predykcji) czy propagacja wsteczna będą w dwóch poniższych plikach.

forward.py – plik implementujący pojedyńczą warstwę w procesie liczenia estymacji (predykcji), jednym z parametrów będzie funkcja aktywacji – będzie można użyc jednej z wielu zdefiniowanych (w tym również funkcji liniowej – czyli brak aktywacji, co zostanie użyte w ostatniej warstwie). W pliku tym będzie również metoda (funkcja) do policzenia kosztu.

backprop.py – plik implementujący pojedyńczą warstwę w procesie propagacji wstecznej, jednym z parametrów będzie funkcja aktywacji (dostarczona zostanie implementacja pochodnej tej funkcji) – będzie można użyć jednej z wielu zdefuniowanych (w tym również funkcji liniowej – czyli brak aktywacji, co zostanie użyte w ostatniej warstwie). W tym pliku będzie również metoda (funkcja) do aktualizacji parametrów.

relu.py – implementacja funkcji ReLU oraz jej pochodnej

sigmoid.py – implementacja funkcji sigmoidalnej oraz jej pochodnej

function\_to\_estimate.py – implementacja funkcji, którą zechemy estymować siecią neuronową