Michał Przysucha
Kacper Radzikowski

WMH – projekt
PB2. Zastosowanie sieci neuronowej w zadaniu aproksymacyjnym

Niniejszy dokument składa się z 3 części:
- rozdział 0: zawiera ustalenie wstępne, wysłane 31.10.2017 – nic tu nie zostało zmienione,
- rozdział 1: zawiera sprawozdanie cząstkowe, wysłane 4.12.2017 – nic tu nie zostało zmienione,
- rozdział 2: zawiera sprawozdanie końcowe – całkowicie nowy rozdział opisujący przeprowadzone doświadczenia.

0. Wstępne ustalenia

Wstępne ustalenia:
1. Język programowania: Python
2. Dane uczące i testujące: wygenerowane z nietrywialnej funkcji $\mathbb{R}^2 \rightarrow \mathbb{R}$
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. Sprawozdanie cząstkowe

1.1 Oznaczenia użyte w opisie oraz w kodzie źródłowym

L- liczba wartsw w sieci neuronowej
l- indeks warstwy w sieci neuronowej
$n^{[l]}$- liczba węzłów w l-tej warstwie sieci neuronowej, w szczególności:
$n^{[0]}$- liczba węzłów w 0-wej warstwie (warstwie wejściowej) sieci neuronowej (liczba cech danych trenujących – w naszym zadaniu $n^{[0]}=2$)
m- liczba danych trenujących
i- indeks danych trenujących (np. i-ty przykład w zbiorze trenującym)
$a^{(i)[l]}$- 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:
$A^{[l]}$- macierz wyjść l-tej warstwy dla wszystkich danych trenujących
$x^{(i)}=a^{(i)[0]}$- wartość na wejściu sieci neuronowej dla i-tej danej trenującej
$X=A^{[0]}$- 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)
$z^{[i][l]}$- 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:
$Z^{[l]}$- macierz wyjść wielomianiu w l-tej warstwie sieci neuronowej (przed funkcją aktywacji) dla wszystkich danych trenujących
$g^{[l]}(Z)$- funkcja aktywacji w l-tej warstwie ($A^{[l]}=g^{[l]}(Z^{[l]})$)
$W^{[l]}, b^{[l]}$- odpowiednio: wagi i wyraz wolny (bias) w l-tej warstwie sieci neuronowej (współczynniki wielomianu wykorzystane do policzenia $Z^{[l]}$, konkretnie: $Z^{[l]}=W^{[l]}\times A^{[l-1]}+b^{[l]}$)
$y^{(i)}=a^{(i)[L]}$- wyjście L-tej warstwy (wyjściowej) sieci neuronowej dla i-tej danej trenującej (jest to predykcja dla danych wartości współczynników W, b 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:
$\hat{Y}=A^{[L]}$ - wartość na wyjściu L-tej warstwy (wyjściowej) sieci neuronowej (predykcja), jest to macierz wyjść wszystkich danych trenujących.
$dW^{[l]}, db^{[l]}, dA^{[l]}, dZ^{[l]}$ - 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)`).

$A^{[l]}$, $Z^{[l]}$- 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:$(n^{[l]}, m)$.

$W^{[l]}:(n^{[l]}, n^{[l-1]})$- 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: W będzie słownikiem / tablicą asocjacyjną, gdzie kluczem będzie l (numer warstwy sieci neuronowej), a wartością macierz $W^{[l]}:(n^{[l]}, n^{[l-1]})$.

$b^{[l]}:(n^{[l]}, 1)$- wektor wyrazów wolnych, liczba wierszy jest taka sama jak liczba węzłów w l-tej warstwie (wymiar wektora wyjściowego z l-tej warstwy)
Implementacja: b będzie słownikiem / tablicą asocjacyjną, gdzie kluczem będzie l (numer warstwy sieci neuronowej), a wartością wektor $b^{[l]}:(n^{[l]}, 1)$.

$X = a^{[0]}$- w niniejszym zadaniu pojedynczy przykład danej trenującej jest wektorem dwuwymiarowym (dwuelementowym), natomiast X będzie macierzą, gdzie wiersze to cechy zmiennej wejściowej, a kolumny to kolejne przykłady danych trenujących, czyli będzie to macierz o wymiarach: $(2, m)$.

$\hat{Y} = A^{[L]}$- macierz o 1 wierszu (dla pojedynczej danej wejściowej, wyjście sieci neuronowej w naszym zadaniu to liczba rzeczywista) oraz o m kolumnach, czyli: $(1, m)$.

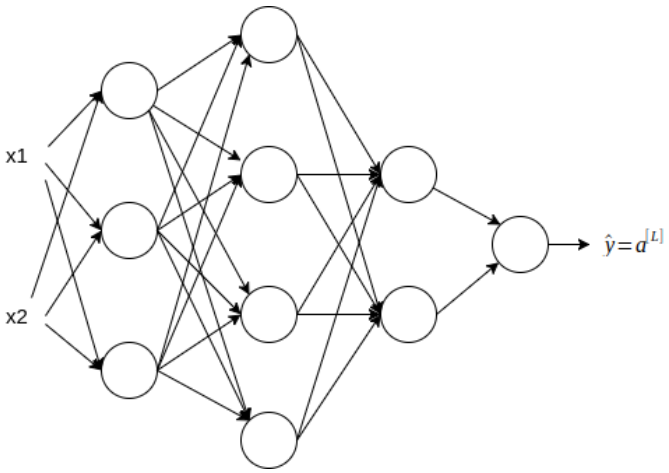
$Z^{[1]} : (n^{[1]}, m)$- wyjście wielomianu w l-tej warstwie (przed funkcją aktywacji) dla wszystkich danych trenujących, liczba wierszy jest taka sama jak liczba węzłów w l-tej warstwie (wymiar wektora wyjściowego z l-tej warstwy dla pojedynczej danej trenującej lub testującej), a liczba kolumn jest taka sama jak liczba danych trenujących lub testujących: $(n^{[l]}, m)$.

$A^{[1]} : (n^{[1]}, m)$- 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: $(n^{[l]}, m)$.

$dW^{[l]}, db^{[l]}, dA^{[l]}, dZ^{[l]}$ - wymiary gradientów, są takie same jak wymiary odpowiadających im macierzy / wektorów: $W^{[l]}, b^{[l]}, A^{[l]}, Z^{[l]}$, czyli np. wymiar macierzy $dW^{[l]}$ jest dokładnie taki sam jak macierzy $W^{[l]}$ itd.

1.3. Przykład

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

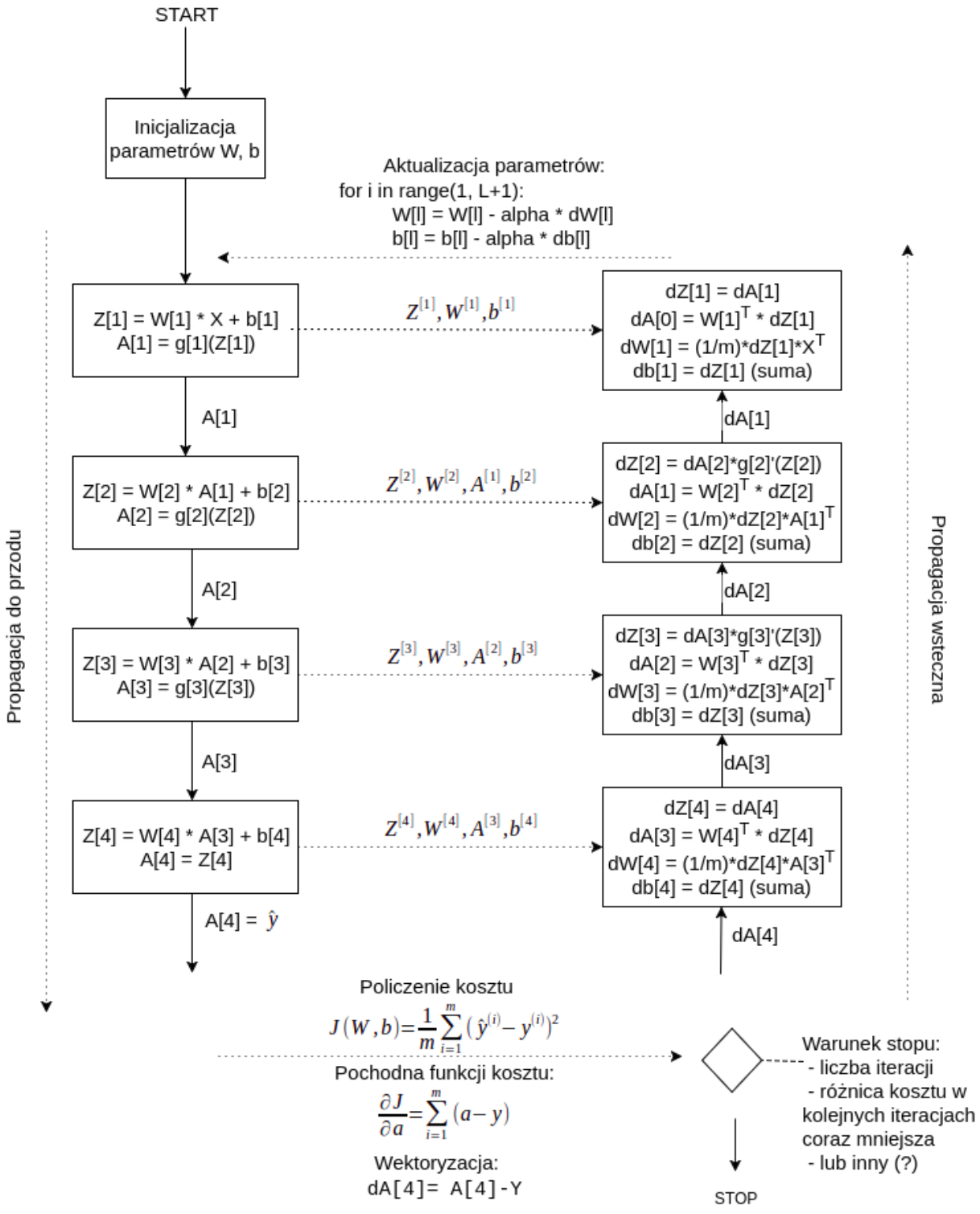


W powyższej sieci neuronowej mamy 3 warstwy ukryte oraz warstę wyjściową, zatem L = 4 (warstwy wejściowej nie liczymy). W poszczególnych warstwach mamy liczbę węzłów:
$n^{[1]} = 3$
$n^{[2]} = 4$
$n^{[3]} = 2$
$n^{[4]} = 1$
Zmienna wejściowa jest dwuwymiarowa (dwuelementowy wektor), zatem $n^{[0]} = 2$.
Wymiary macierzy ze współczynnikami:

$W^{[1]}:(3,2)$
$W^{[2]}:(4,3)$
$W^{[3]}:(2,4)$
$W^{[4]}:(1,2)$
Wymiary wektorów z wyrazami wolnymi:
$b^{[1]}:(3,1)$
$b^{[2]}:(4,1)$
$b^{[3]}:(2,1)$
$b^{[4]}:(1,1)$

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 – α. 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 α. Będziemy monitorować proces uczenia (więcej w nastepnym punkcie 1.5) i na bieżąco decydować, czy przerwać proces z powodu zbyt małej / dużej wartości współczynnika α.

Poniższe kroki opisują proces uczenia na zbiorze trenującym:

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

2. Następnie wykorzystując współczynniki przechodzimy przez kolejne warstwy sieci neuronowej, aż obliczymy $\hat{Y}$, czyli estymację (predykcję) Y. 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 dW oraz db dla każdej z warstw.

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

Po aktualizcji parametrów ponownie wykorzystamy sieć neuronową i zaktualizowane współczynniki to policzenia estymacji (predykcji), czyli do policzenia $\hat{Y}$. 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 R^2).

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 pojedyncza 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

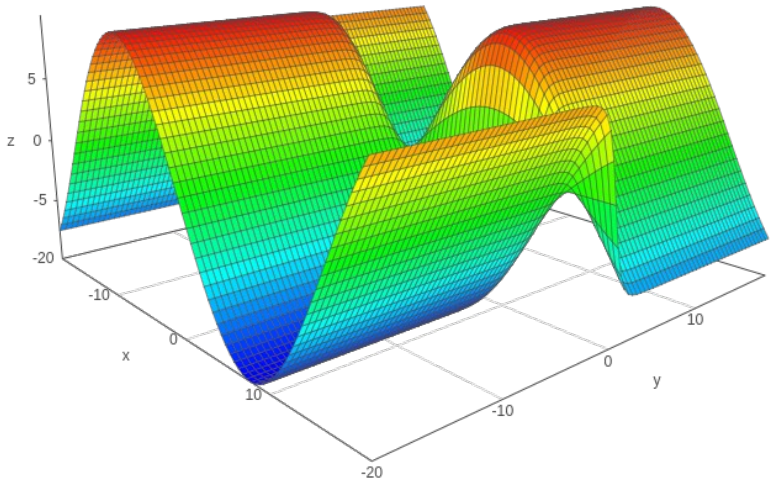
function_to_estimate.py – implementacja funkcji, którą zechemy estymować siecią neuronową

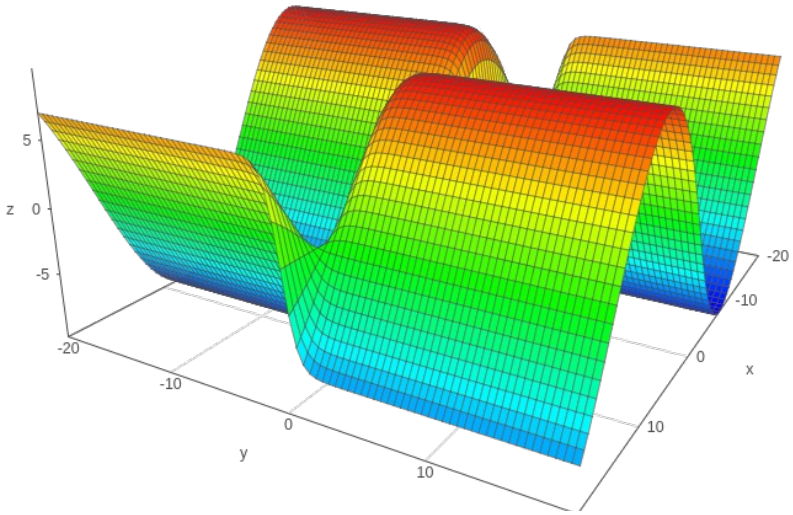
2. Sprawozdanie końcowe

2.1. Opis problemu do rozwiązania

Za pomocą sieci neuronowej estymujemy funckję $\mathbb{R}^2 \rightarrow \mathbb{R}$. Przyjęliśmy następującą funkcję:
10 * sin(X/5) * tanh(Y)

Jej wykres wygląda nastepująco:





2.1. Opis działania programu

Architekturę sieci zapisujemy w nastepujący przykładowy sposób:

```
layers = [2, 2, 1]
```

Oznacza to jedną warstwę ukrytą (drugi element tablicy) z 2 neuronami i jedną wartwą wyjściową (ostatni element tablicy) z 1 neuronem. Pierwsze element tablicy to wejście sieci – wpisane zostało, aby uprościć implementację. Nie jest liczony jako oddzielna warstwa, nie ma dla niego współczynników (wielomianu), ani tym bardziej funkcji aktywacji. Zawsze pierwszym elementem będzie liczba 2, bo nasz problem to estymacja funkcji, której wejście to 2-elementowy wektor ($\mathbb{R}^2$).

Zatem liczba warstw zawsze będzie wyliczana w sposób:

```
L = len(layers) - 1  # number of layers - input layer doesn't count
```

Każde budowanie modelu wymaga ustawienia dodatkowych hiperparametrów, np.:

```
number_of_iterations = 10000
learning_rate = 0.05
```

Oznacza to, że wykonamy 10000 iteracji ze współczynnikiem uczenia 0.05.
Program wyświetla koszt na zbiorze trenującym i na zbiorze testowym co każde 50 iteracji, np.:

```
Train history:
Iteration: 50   Training cost: 41.84100 Testing cost: 42.32823
Iteration: 100   Training cost: 41.83035 Testing cost: 42.29750
Iteration: 150   Training cost: 41.81363 Testing cost: 42.26608
Iteration: 200   Training cost: 41.74670 Testing cost: 42.21037
```

Po wykonaniu 10000 iteracji program zapyta nas czy chcemy konktynuować.
Możemy podać dodatkową liczbę iteracji:

```
Iteration: 9950   Training cost: 26.29705 Testing cost: 30.25534
Iteration: 10000   Training cost: 26.29705 Testing cost: 30.25534
Would you like to continue? [y/n]: y
How many addition iterations?10000
```

I tak do momentu, w którym zdecydujemy się przerwać.

Dodatkowo program rysuje wykres funkcji kosztu, który może być odkładany w oddzielnym pliku po wybranej liczbie iteracji, np.:

```
save_cost_plot_after_epoch = list([50, 100, 1000])
```

Otrzymamy w oddzielnych plikach wykresy funkcji kosztu po 50, 100 oraz 1000 iteracji.
Dodatkowo odłożone zostają wykresy funkcji kosztu na koniec wszystkich iteracji, nawet jeśli zdecydujemy się kontynuować trening. W powyższym przykładzie po 10000 iteracji, jeśli zdecydujemy się na kolejne 10000 iteracji, otrzymamy dodatkowe pliki z wykresem funkcji kosztu po 10000 oraz po 20000 iteracjach. Wyświetlanie kosztu oraz wykresy funkcji kosztu pomagają w podjęciu decyzji, czy trenujemy dalej.

Jeśli zdecydujemy się zakończyć program wykonuje „sanity check” oraz wyświetlenie metryk.

Na koniec model wraz w obrazami funkcji kosztu jest zapisywany z kolejnym numerkiem w nazwach plików.

2.1.1. Sanity check

Polega na wylosowaniu kilku punktów i sprawdzeniu jakie wartości daje funkcja oraz jakie predykcje daje wytrenowany model. Np.:

```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[ 3.39039669]]
```

Jest to przykład dla jednego punktu. Wykonujemy taki test dla 4 punktów z każdej ćwiartki zbioru wejściowego (X[0] dodatnie oraz X[1] dodatnie, X[0] ujemne oraz X[1] dodatnie, itd.)

2.1.2. Wyświetlenie metryk

Wyświetlamy błąd średniokwadratowy, współczynnik determinacji oraz korelację Pearsona i Spearmana, np.:

```
Regression metrics:
MSE: 0.17585994477
R2:  0.318434275659
Pearson: 0.544, p-value: 0.000
Spearman: 0.469, p-value: 0.000
```

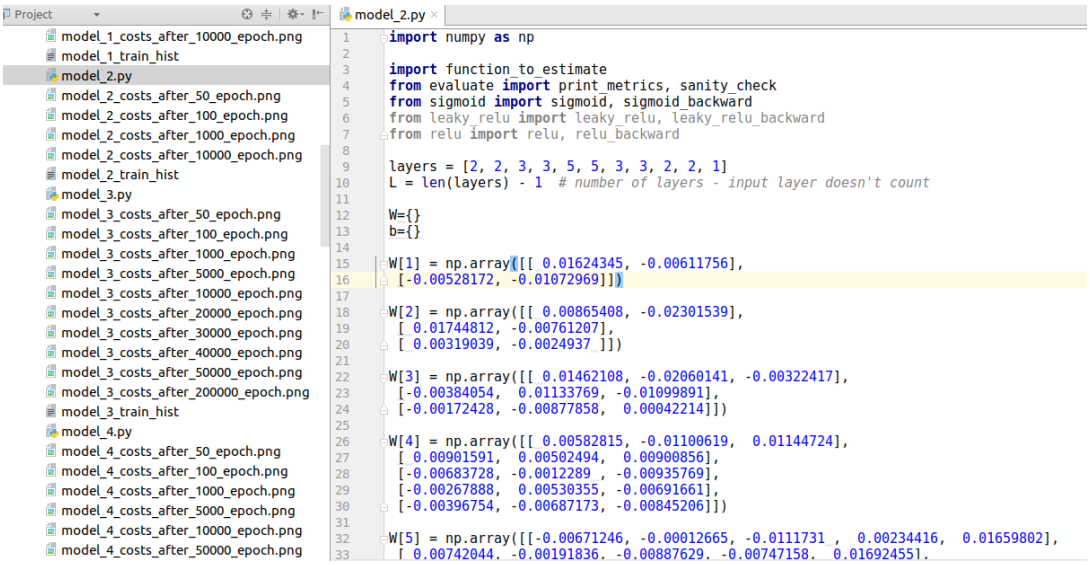
2.1.3. Zapisanie modelu

Na koniec model jest zapisywany w formie pliku Python, który można uruchomić. Uruchomienie spowoduje zbudowanie sieci w wytrenowanymi współczynnikami oraz wykonanie na niej powyższych dwóch kroków (sanity check oraz wyświetlenie metryk).

Sanity check zawsze zwróci te same wyniki, bo ustawiony jest na sztywno ziarno funkcji losującej w funkcji `sanity_check`:

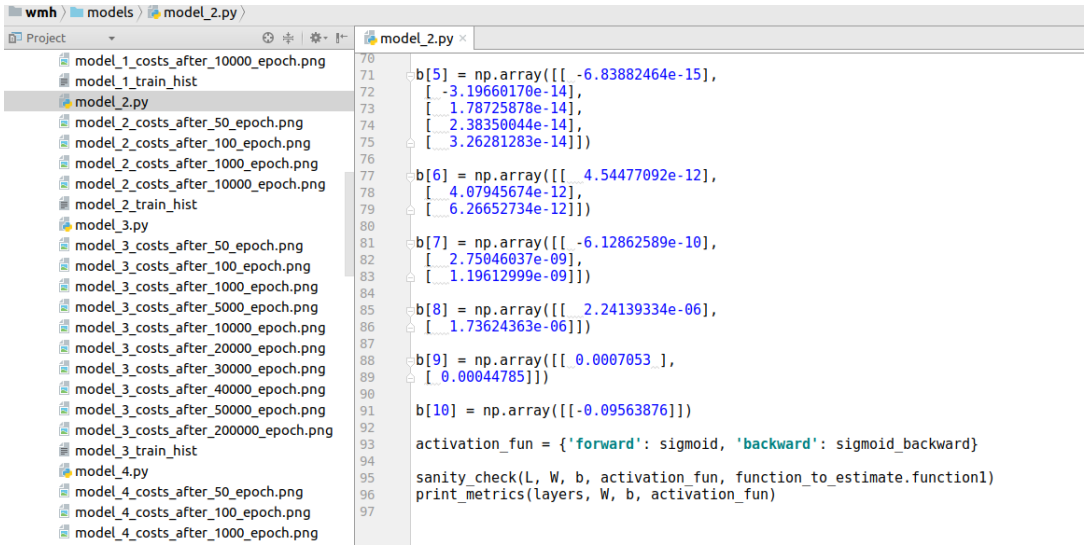
```
np.random.seed(1)
```

Przykładowy model:

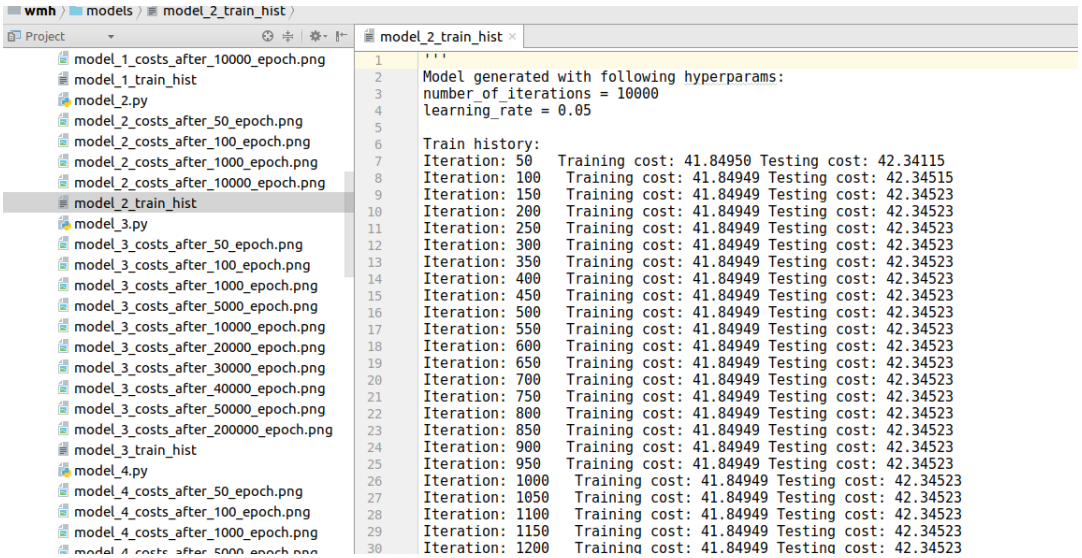


Model został zapisany w pliku model_2.py, gdzie widzimy ustawienie liczby warstw oraz ustawione współczynniki dla każdej z warstw.

Na dole tego samego pliku (model_2.py) została zapisana również funkcja aktywacji oraz uruchomienie sanity check i wyświetlenia metryk:



Dodatkowo zapisana została historia treningu modelu:



A także wykresy funkcji kosztu po wybranej liczbie iteracji – w powyższym zrzucie ekranu widać 4 pliki z wykresami funkcji kosztu dla modelu 2:

- model_2_costs_after_50_epoch.png
- model_2_costs_after_100_epoch.png
- model_2_costs_after_1000_epoch.png
- model_2_costs_after_10000_epoch.png

2.2. Modele

Rozdział opisuje wytrenowane modele wraz z metrykami, funkcjami kosztu oraz wnioskami. Zawiera 12 podpunktów, bo wypróbowaliśmy 12 różnych architektur sieci neuronowej:
- 4 architektury z funkcjami aktywacji Sigmoid
- 4 architektury z funkcjami aktywacji Leaky ReLU
- 4 architektury z funkcjami aktywacji ReLU

Dla każdej z funkcji aktywacji próbowaliśmy po 4 architektury:

- `[2, 2, 1]` – zaczęliśmy od najprostszej architektury z 1 warstwą ukrytą z 2 neuronami i warstwą wyjściową z jednym neuronem,
- `[2, 2, 3, 3, 5, 5, 3, 3, 2, 2, 1]` – następnie dodaliśmy wiele warstw ukrytych (konkretnie: 9)
- `[2, 20, 20, 1]` – ponieważ powyższa architektura (niezależnie od funkcji aktywacji) dawały słabe wyniki, więc zostawiliśmy 2 warstwy ukryte, ale dodaliśmy liczbę neuronów (konkretnie: po 20 na warstwę),
- `[2, 100, 100, 1]` – j.w., ale jeszcze więcej neuronów w obu warstwach ukrytych (po 100).

2.3.1. Architektura [2, 2, 1], funkcja aktywacji: Sigmoid

Zaczynamy od prostej architektury sieci neuronowej z 2 neuronami w warstwie ukrytej i 1 w warstwie wyjściowej.

Ustawiamy hiperparametry:

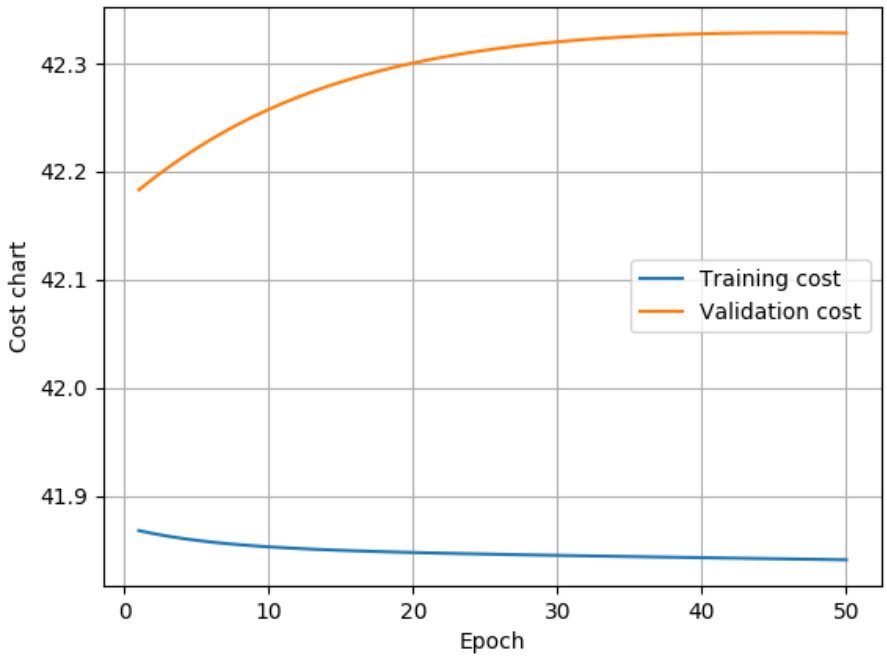
```
number_of_iterations = 10000
learning_rate = 0.05
```

oraz liczbę iteracji, po której otrzymamy wykresy funkcji kosztu:

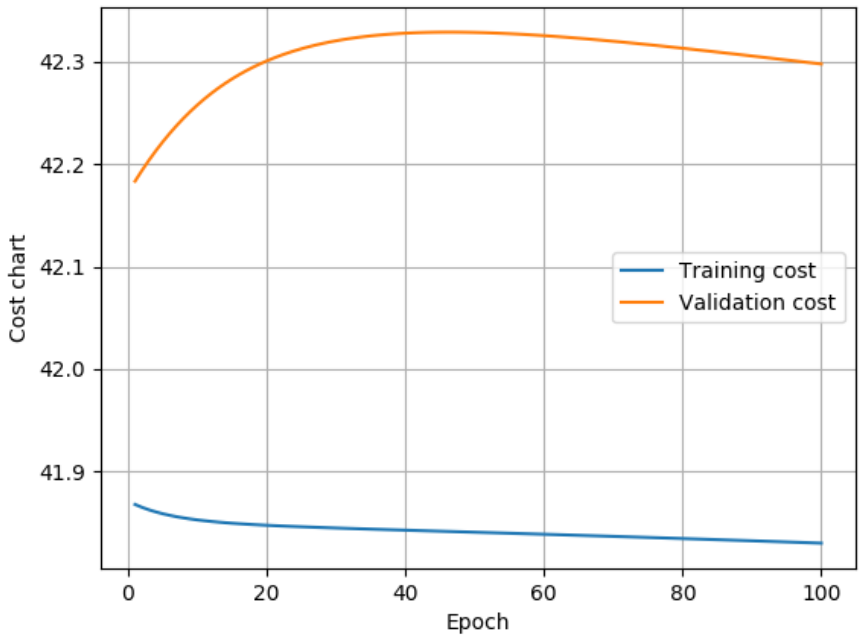
```
save_cost_plot_after_epoch = list([50, 100, 1000])
```

Obserwujemy proces uczenia poprzez wizualizację funkcji kosztu od epoch (liczby przejść przez cały zbiór trenujący).

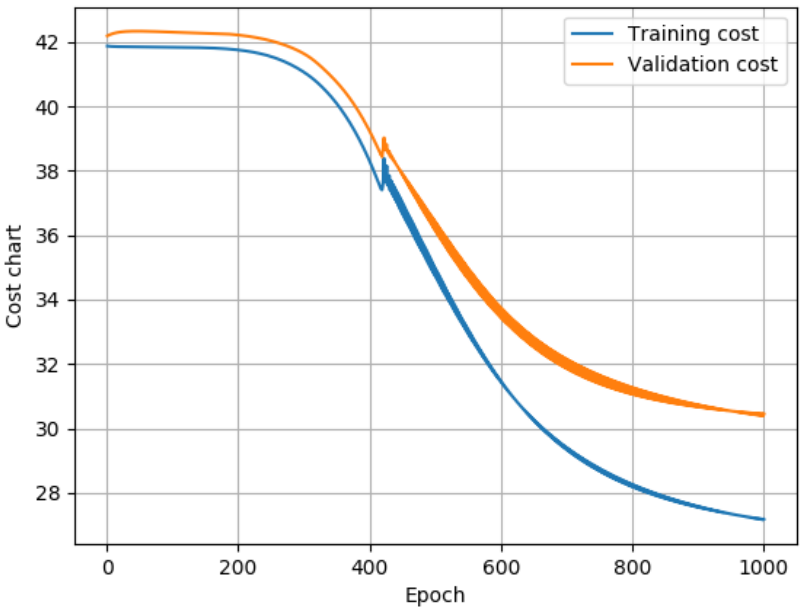
Po 50 iteracjach koszt na zbiorze trenującym nieznacznie spadł, a na zbiorze testowym nawet wzrósł:



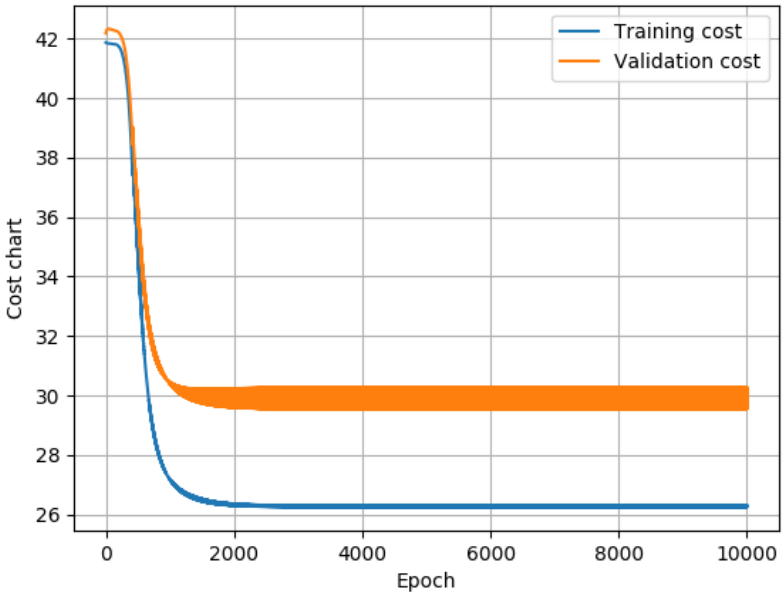
Po 100 iteracjach jest nieznaczna poprawa:



Po 1.000 iteracjach jest już lepiej:



Po 10.000 iteracji mamy ustabilizowany model, niestety z ciągle dużym kosztem:



Na koniec otrzymujemy sanity check oraz metryki:

```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[ 3.39039669]]

X=[[ -2.28749635e-03],
```

```
 [  6.04665145e+00]]
real value:  [-0.00457494]
prediction: [[-5.88394002]]

X=[[ 2.93511782],
 [-1.8467719 ]]
real value:  [-5.26989202]
prediction: [[-6.06727222]]

X=[[-3.72520423],
 [-6.91121454]]
real value:  [ 6.78000488]
prediction: [[ 1.34608285]]

Regression metrics:
MSE: 0.17585994477
R2:  0.311184566732
Pearson: 0.544, p-value: 0.000
Spearman: 0.469, p-value: 0.000
```

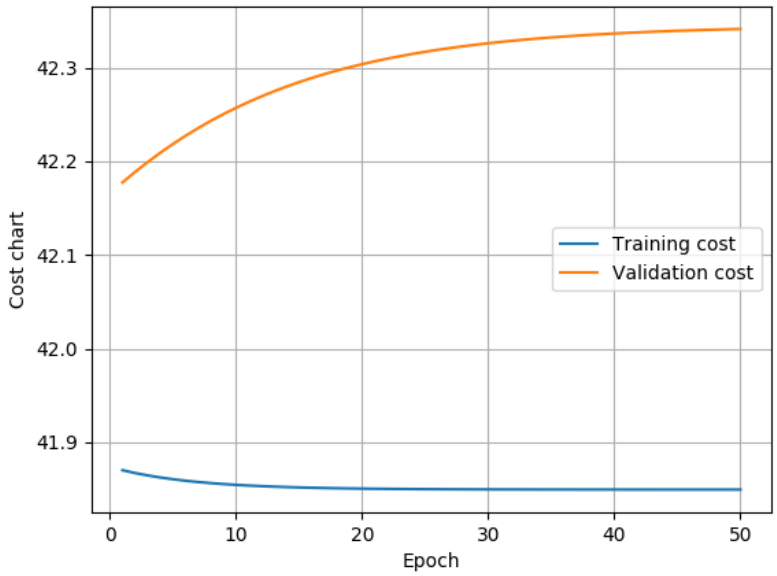
Model słabo działa, dlatego spróbowaliśmy dodać wiele warstw ukrytych.

2.3.2. Architektura [2, 2, 3, 3, 5, 5, 3, 3, 2, 2, 1], funkcja aktywacji: Sigmoid

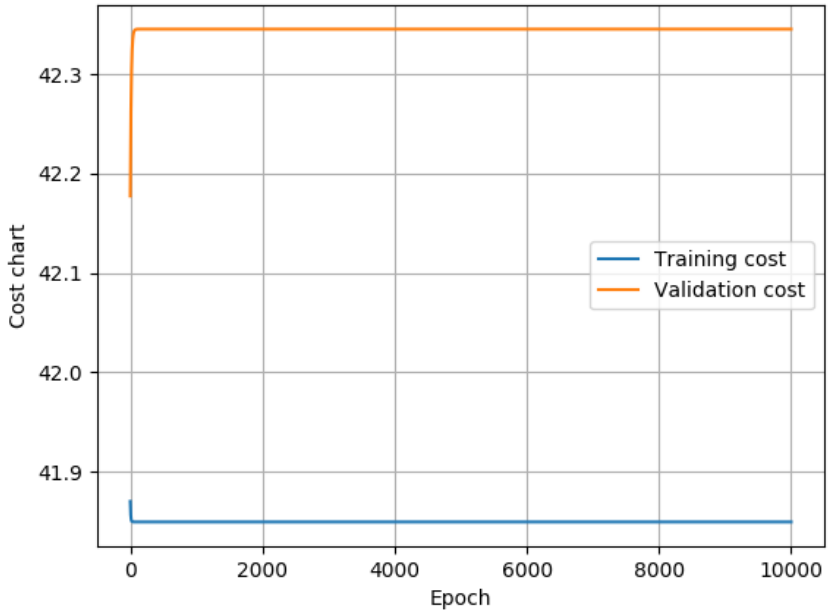
Hiperparametry:

```
number_of_iterations = 10000
learning_rate = 0.05
```

Po 50 iteracjach mamy nieznaczny spadek kosztu na zbiorze trenującym i wzrost na zbiorze testowym:



Niestety kolejne iteracje nie przynoszą żadnego efektu. Ostatecznie kończymy po 10.000 iteracji:

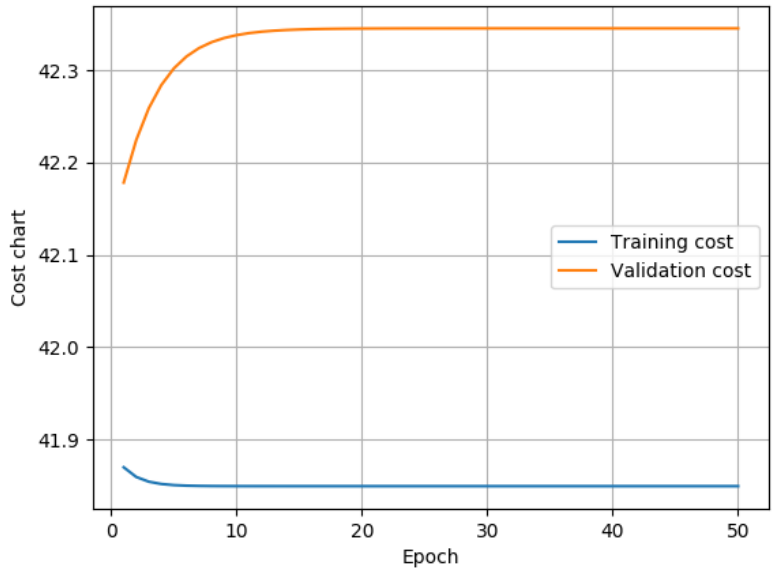


2.3.3. Architektura [2, 20, 20, 1], funkcja aktywacji: Sigmoid

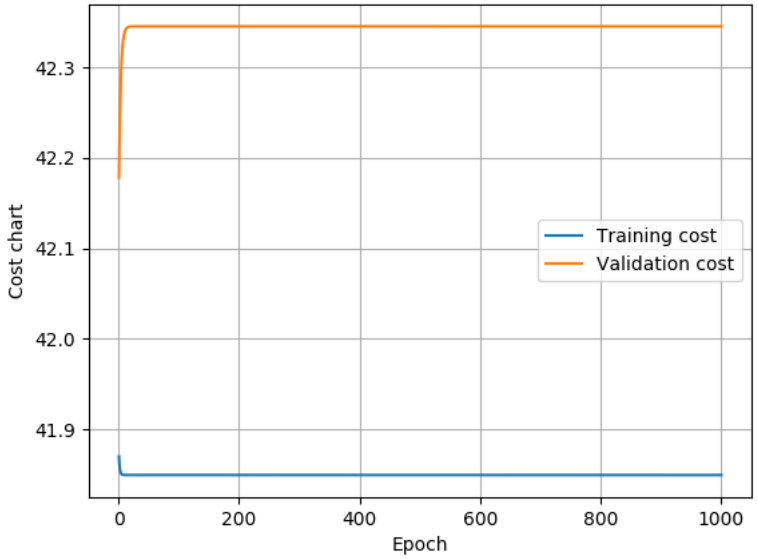
Hiperparametry:

```
number_of_iterations = 10000
learning_rate = 0.05
```

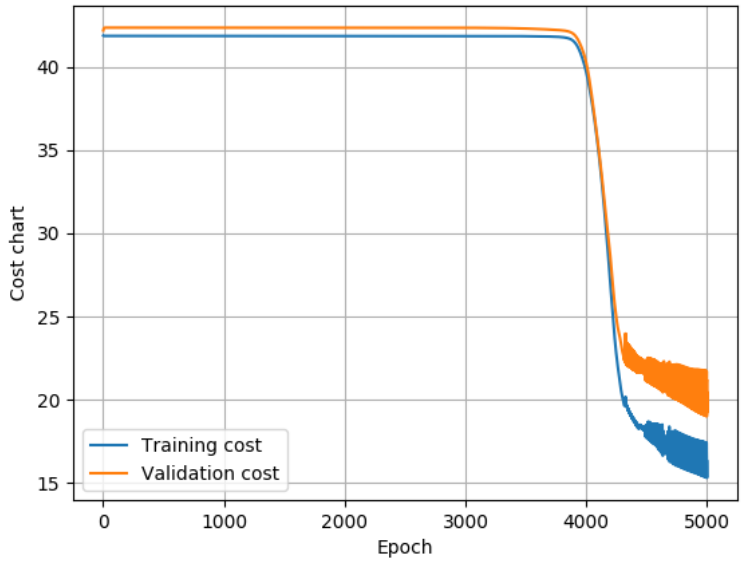
Z początku wykres funkcji kosztu wygląda podobnie, po 50 iteracjach:



Po 1.000 iteracjach:



Jednak po 4.000 iteracji koszty zaczynają spadać:

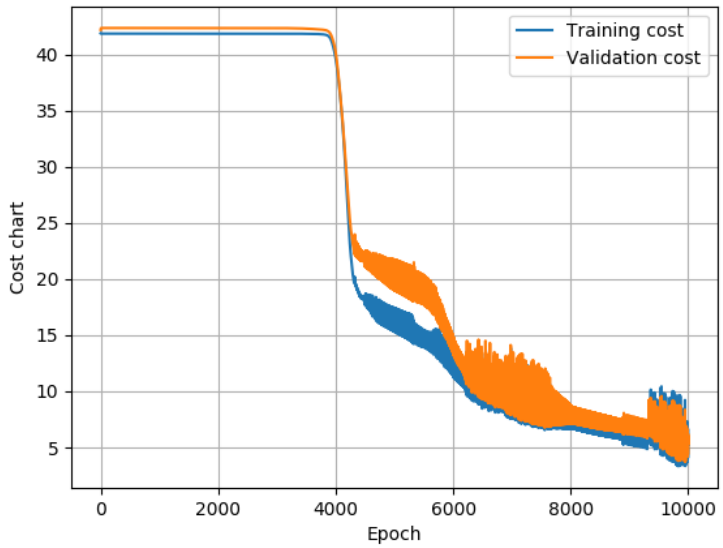


W rzeczywistości wcześniej koszty też spadały, jednak na tyle nieznacznie, że nie widać tego na wykresie. Dopiero po przekroczeniu 4000 koszt zaczął spadać:

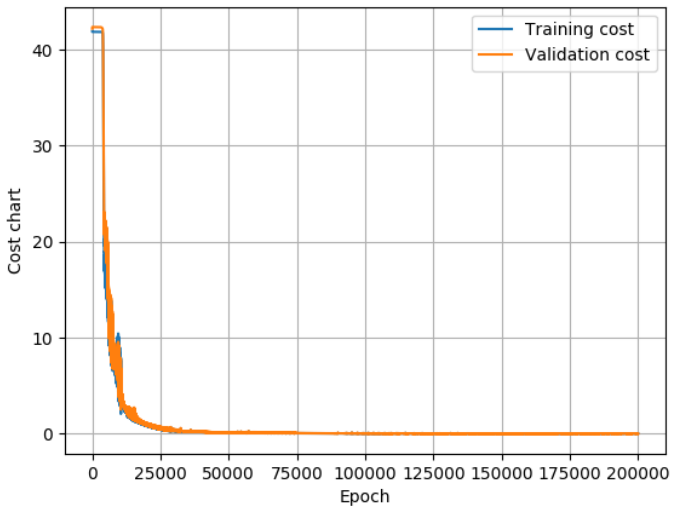
```
Iteration: 3500   Training cost: 41.83181 Testing cost: 42.29184
Iteration: 3550   Training cost: 41.82813 Testing cost: 42.27952
Iteration: 3600   Training cost: 41.82364 Testing cost: 42.26520
Iteration: 3650   Training cost: 41.81800 Testing cost: 42.24906
Iteration: 3700   Training cost: 41.81028 Testing cost: 42.23125
Iteration: 3750   Training cost: 41.79794 Testing cost: 42.21098
Iteration: 3800   Training cost: 41.77330 Testing cost: 42.18265
Iteration: 3850   Training cost: 41.70951 Testing cost: 42.12212
Iteration: 3900   Training cost: 41.49427 Testing cost: 41.92769
Iteration: 3950   Training cost: 40.85476 Testing cost: 41.34320
```

```
Iteration: 4000   Training cost: 39.65350 Testing cost: 40.21631
Iteration: 4050   Training cost: 37.68495 Testing cost: 37.98973
Iteration: 4100   Training cost: 34.99637 Testing cost: 35.20169
Iteration: 4150   Training cost: 31.35753 Testing cost: 31.92327
Iteration: 4200   Training cost: 26.89051 Testing cost: 28.50306
Iteration: 4250   Training cost: 22.62861 Testing cost: 24.84856
Iteration: 4300   Training cost: 20.11676 Testing cost: 22.85406
Iteration: 4350   Training cost: 19.32470 Testing cost: 23.27921
Iteration: 4400   Training cost: 18.64912 Testing cost: 22.82424
Iteration: 4450   Training cost: 18.33623 Testing cost: 22.49508
Iteration: 4500   Training cost: 18.07904 Testing cost: 22.21484
```

Po wykonaniu 10.000 iteracji otrzymujemy ciekawy wykres funkcji kosztu:



Decydujemy się kontynuować trenowanie modelu finalnie po 200.000 iteracji otrzymując:



Jednocześnie dostając obiecujący sanity check, wysoki współczynnik determinacji, niski błąd średniokwadratowy i wysoką korelacją:

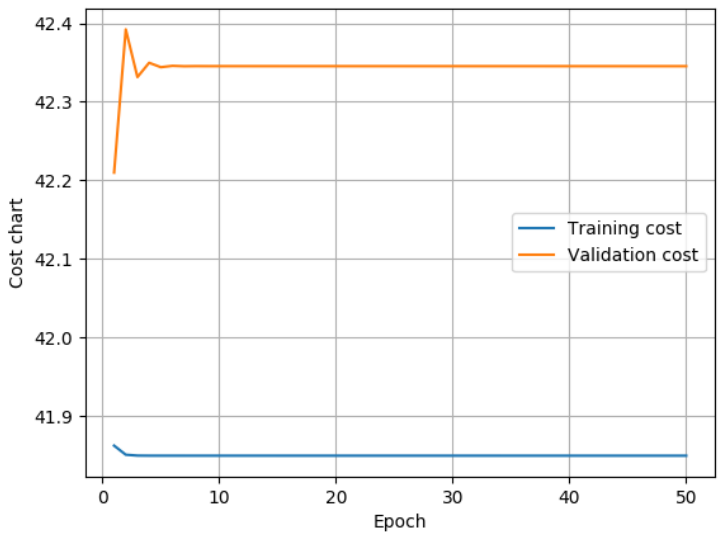
```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[ 9.95851896]]

X=[[ -2.28749635e-03],
 [  6.04665145e+00]]
real value:  [-0.00457494]
prediction: [[ 0.16490052]]

X=[[ 2.93511782],
 [-1.8467719 ]]
real value:  [-5.26989202]
prediction: [[-5.10390514]]

X=[[-3.72520423],
 [-6.91121454]]
real value:  [ 6.78000488]
prediction: [[ 6.92093252]]

Regression metrics:
MSE: 0.000149555575082
R2:  0.99939999844
Pearson: 1.000, p-value: 0.000
Spearman: 0.999, p-value: 0.000
```

Model 3 na razie wygrywa.

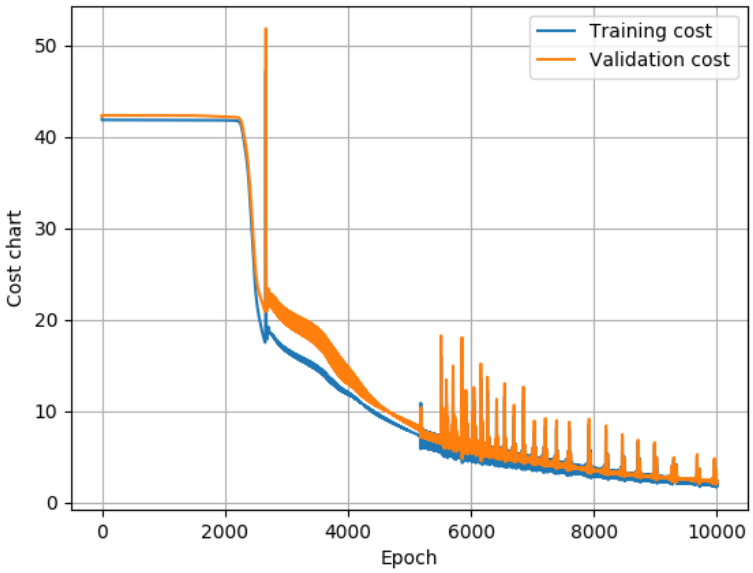
2.3.4. Architektura [2, 100, 100, 1], funkcja aktywacji: Sigmoid

```
number_of_iterations = 10000
learning_rate = 0.05
```

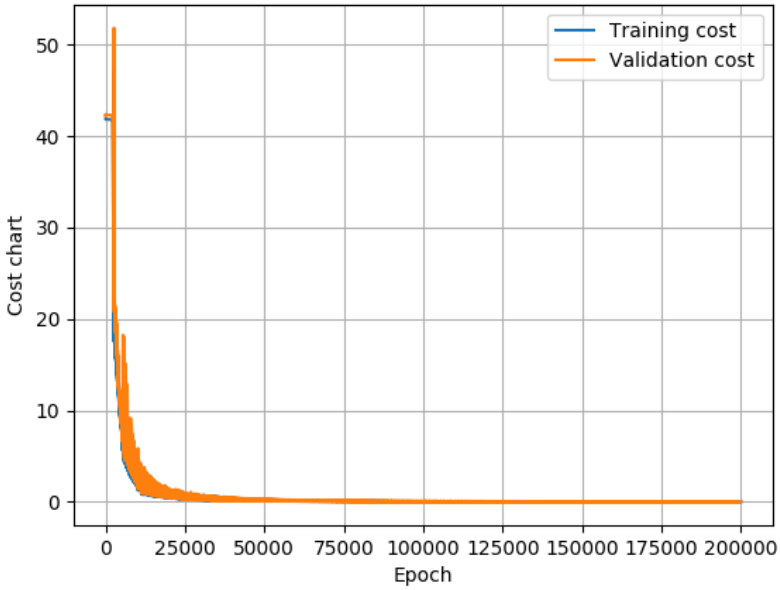
Podobny start:



Po 10.000 iteracji mamy:



Warto kontynuować. Ostatecznie po 200.000 iteracji mamy:



```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[ 9.9704696]]

X=[[ -2.28749635e-03],
 [  6.04665145e+00]]
real value:  [-0.00457494]
prediction: [[-0.00502085]]

X=[[ 2.93511782],
 [-1.8467719 ]]
real value:  [-5.26989202]
prediction: [[-5.33240184]]

X=[[-3.72520423],
 [-6.91121454]]
real value:  [ 6.78000488]
prediction: [[ 6.80059257]]

Regression metrics:
MSE: 5.84025867309e-05
R2:  0.999765782618
Pearson: 1.000, p-value: 0.000
Spearman: 0.999, p-value: 0.000
```

Model 4 daje podobne wyniki co Model 3. Mamy odrobinkę niższy błąd średniokwadratowy i odrobinkę wyższy współczynnik determinacji. Natomiast model jest bardziej złożony. Który wybrać? Największa macierz to macierz współczynników dla warstwy drugiej, ma ona wymiary: 100x100. Nie jest to dużo dla obecnych komputerów. Wybieramy Model 4.

Dodatkowo w Sanity check dla punktu drugiego, gdzie wynik powinien być bliski zera sieć neuronowa z Modelem 4 daje dobrą predykcję, podczas gdy z Modelem 3 jest tam nieznaczna różnica.

Być może gdyby zwiększyć liczbę neuronów w warstwie 2 i 3 otrzymalibyśmy jeszcze lepsze wyniki, ale te są bardzo satysfakcjonujące. Z uwagi na fakt, iż treninng jest czasochłonny nie decydujemy się na modele z większą liczbą neuronów w warstwach 2 i 3. Nie chodzi tutaj o nasz czas podczas wykonywania tego projektu, a o hipotetyczne przełożenie tego problemu na problem bardziej praktyczny. Ponieważ rzeczywistość się ciągle zmienia modele powinny być trenowane cały czas, a następnie porównywane ze starszymi modelami (dla nowszych danych). Zatem przy takich wysokich wynikach nie miałoby sensu znaczące wydłużanie czasu treningu.

Dalej próbowaliśmy wykonać modele dla analogicznych architektur, ale z innymi funkcjami aktywacji w warstwach ukrytych (warstwa wyjściowa ciągle nie ma funkcji aktywacji).

2.3.5. Architektura [2, 2, 1], funkcja aktywacji: Leaky ReLU

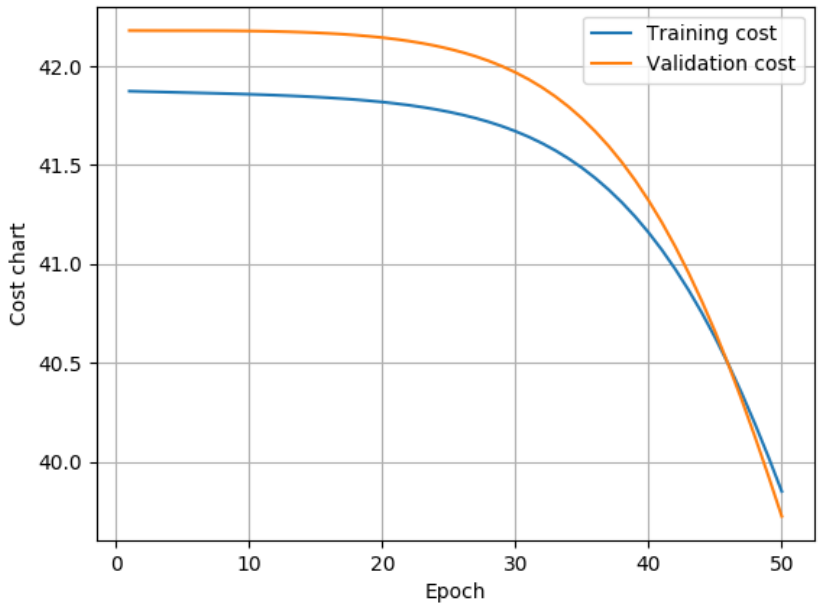
Hiperparametry:

```
number_of_iterations = 10000
learning_rate = 0.05
```

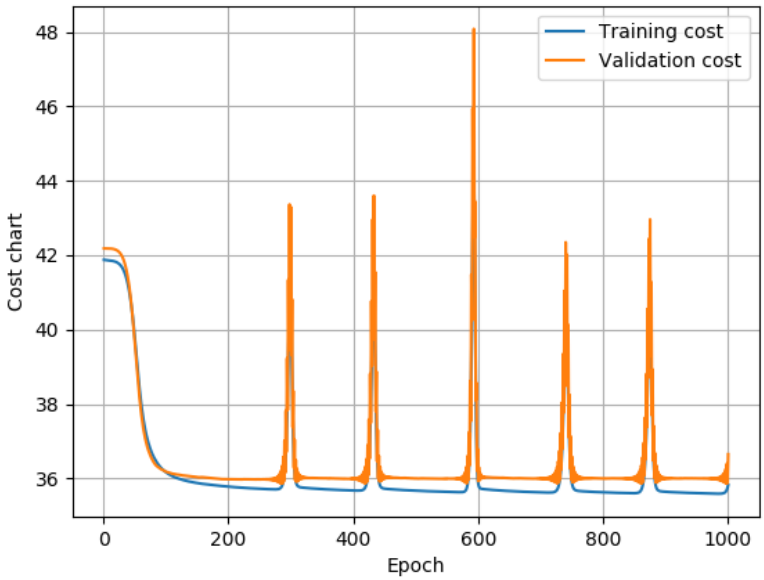
Dla tak ustawionych hiperparametrów funkcja kosztu osiągała ogromne wartości, finalnie osiągając NaN. Zmniejszyliśmy współczynnik uczenia:

```
number_of_iterations = 10000
learning_rate = 0.01
```

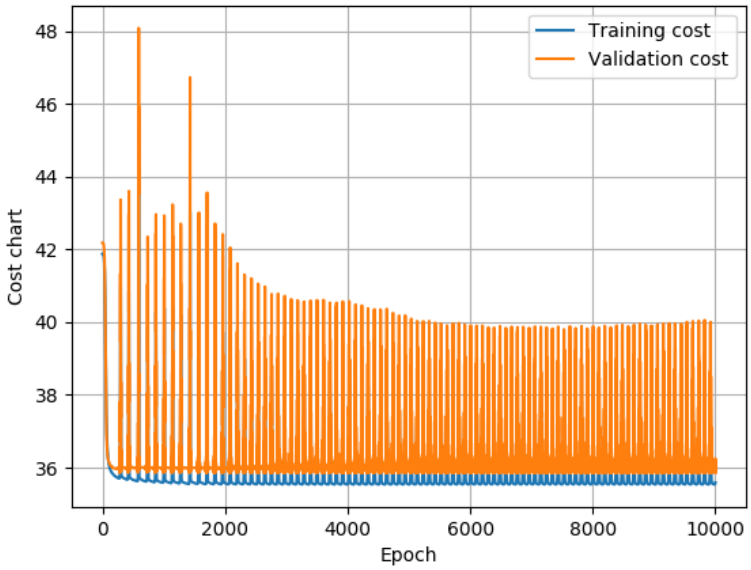
Funkcja kosztu po 50 iteracjach zapowiada się ciekawie:



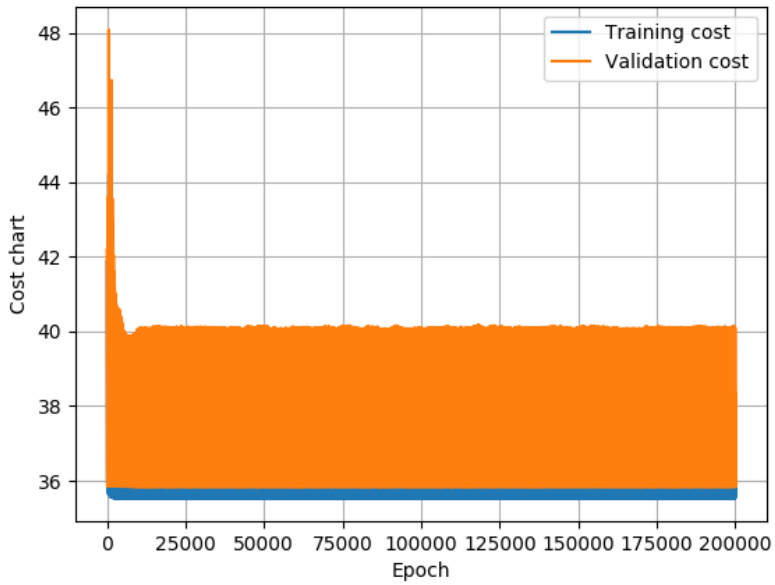
Dalej już jest gorzej. Po 1.000:



Oraz po 10.000:



Zostawiamy na dłużej (200.000 iteracji), ale nie ma żadnych efektów:



Wyniki są beznadziejne:

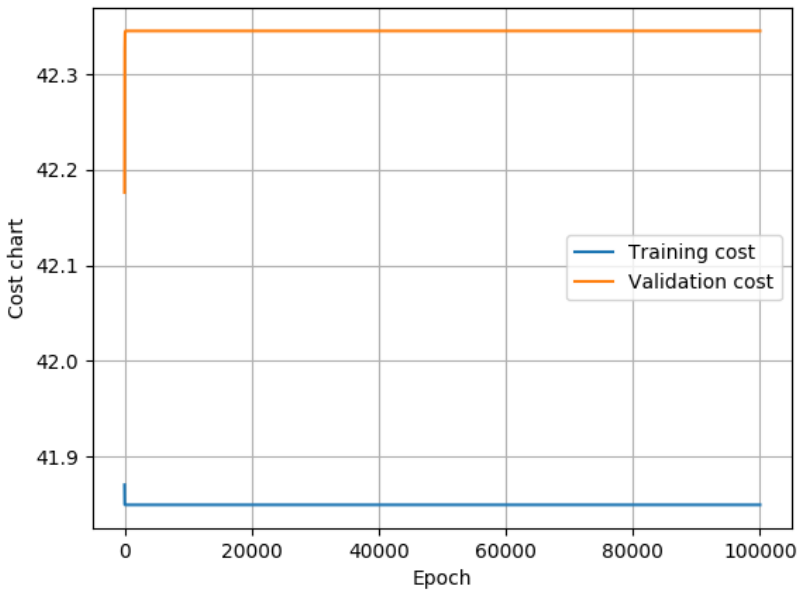
```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[ 5.98279698]]

X=[[ -2.28749635e-03],
 [  6.04665145e+00]]
```

```
real value:  [-0.00457494]
prediction: [[ 0.04110617]]

X=[[ 2.93511782],
 [-1.8467719 ]]
real value:  [-5.26989202]
prediction: [[-0.52092419]]

X=[[-3.72520423],
 [-6.91121454]]
real value:  [ 6.78000488]
prediction: [[-0.46414408]]

Regression metrics:
MSE: 0.232919659734
R2:  0.207993036138
Pearson: 0.317, p-value: 0.000
Spearman: 0.258, p-value: 0.001
```

2.3.6. Architektura [2, 2, 3, 3, 5, 5, 3, 3, 2, 2, 1], funkcja aktywacji: Leaky ReLU

```
number_of_iterations = 10000
learning_rate = 0.05
```

Doszliśmy do 100.000 iteracji, wynik jeszcze gorszy niż poprzedni:

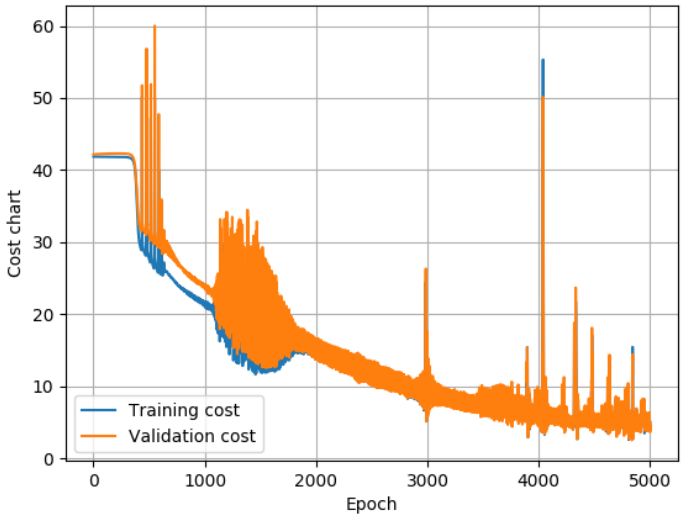


2.3.7. Architektura [2, 20, 20, 1], funkcja aktywacji: Leaky ReLU

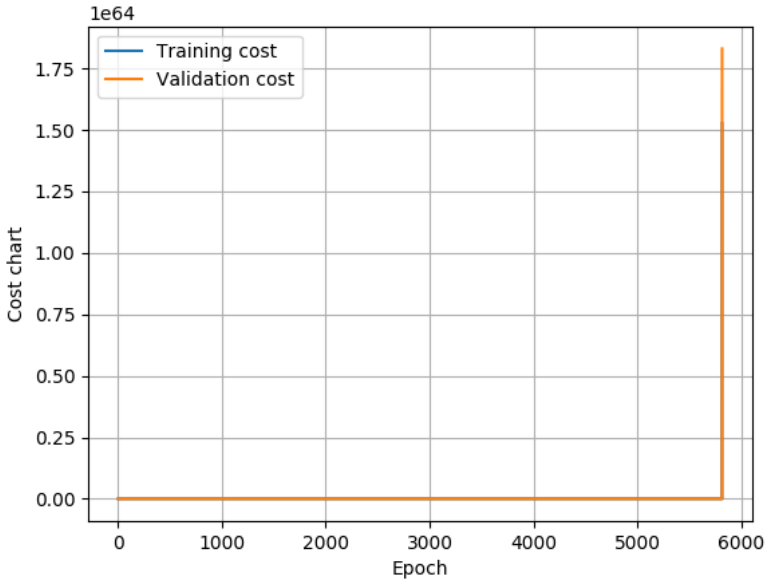
Podobnie jak w Modelu 5, tutaj również musieliśmy użyć mniejszy współczynnik uczenia:

```
number_of_iterations = 10000
learning_rate = 0.01
```

Z początku ciekawie (po 5.000 iteracjach):



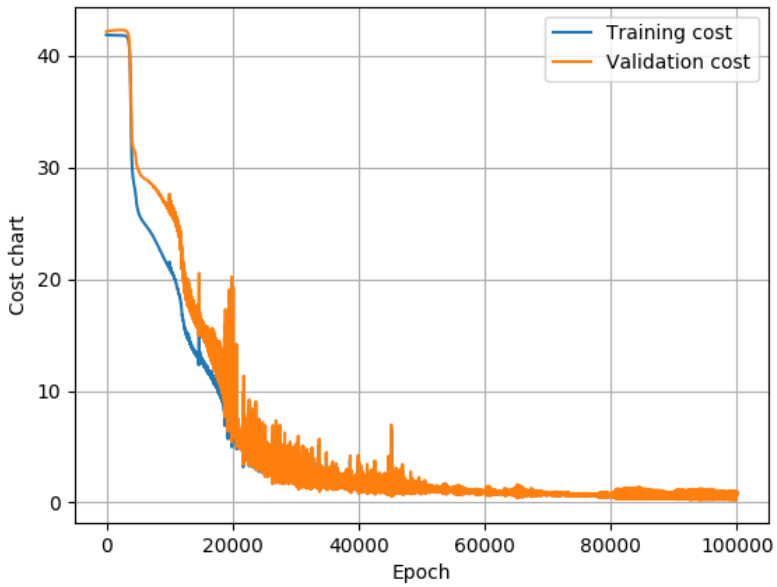
Później już koszt otrzymuje ogromne wartości:



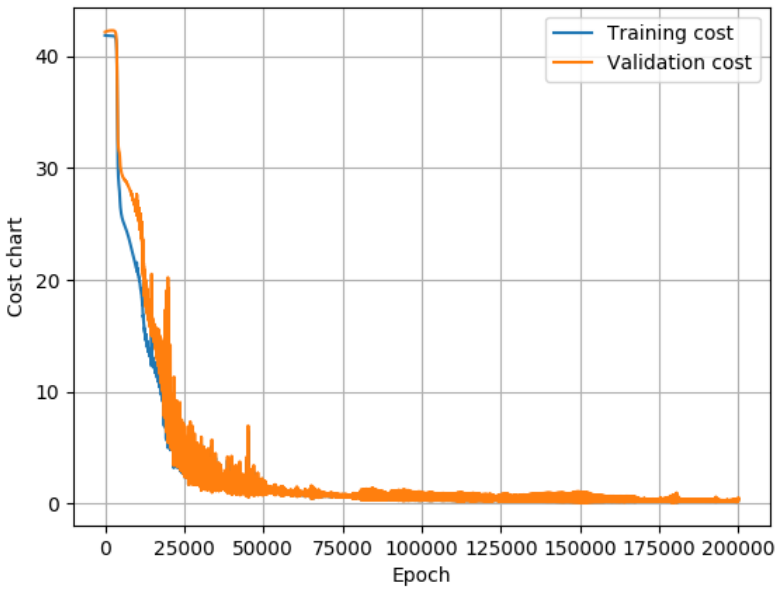
Jeszcze jedna próba dla jeszcze mniejszego współczynnika uczenia:

```
number_of_iterations = 10000
learning_rate = 0.001
```

Po 100000 wygląda ciekawe:



Zostawiamy na kolejne tyle, aby po 200.000 iteracji otrzymać:



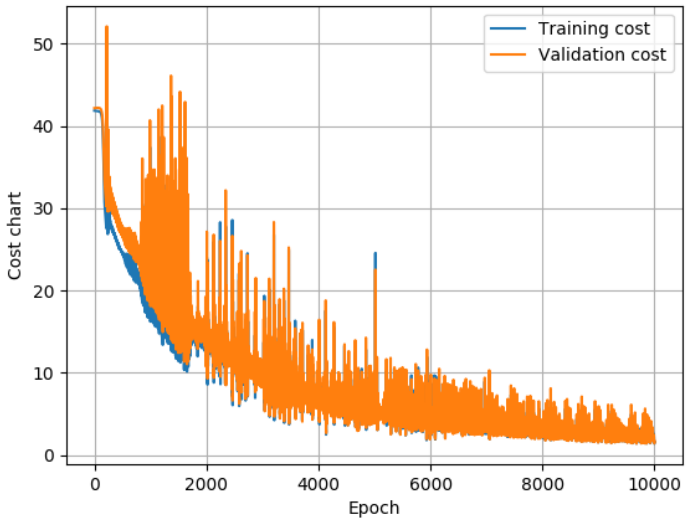
```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[ 9.23415313]]

X=[[ -2.28749635e-03],
 [  6.04665145e+00]]
real value:  [-0.00457494]
prediction: [[-0.30811113]]

X=[[ 2.93511782],
 [-1.8467719 ]]
real value:  [-5.26989202]
prediction: [[-4.71805241]]

X=[[-3.72520423],
 [-6.91121454]]
real value:  [ 6.78000488]
prediction: [[ 6.82713957]]

Regression metrics:
MSE: 0.0024805885721
R2:  0.990221575021
Pearson: 0.996, p-value: 0.000
Spearman: 0.992, p-value: 0.000
```

Wyniki są ok, ale Model 4 był lepszy.

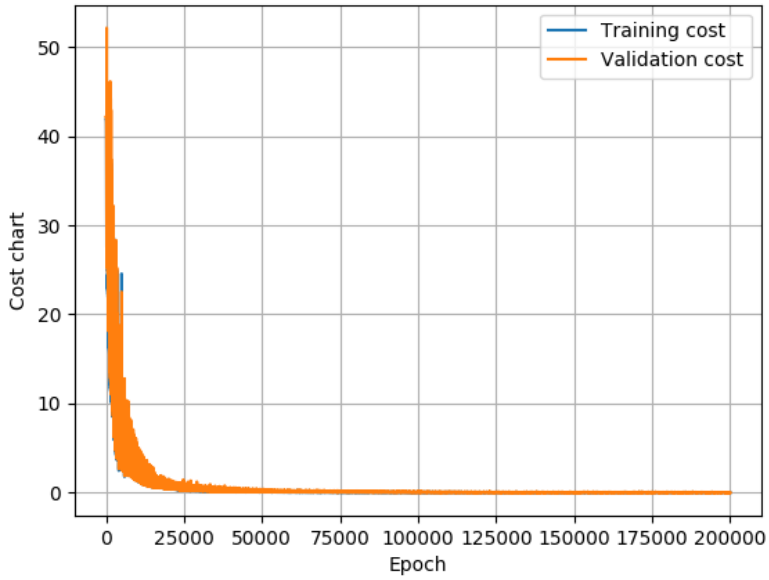
2.3.8. Architektura [2, 100, 100, 1], funkcja aktywacji: Leaky ReLU

```
number_of_iterations = 10000
learning_rate = 0.01
```

Po 10.000 iteracji wygląda obiecująco:



Kontynuujemy, aby po 200.000 iteracji osiągnąć:



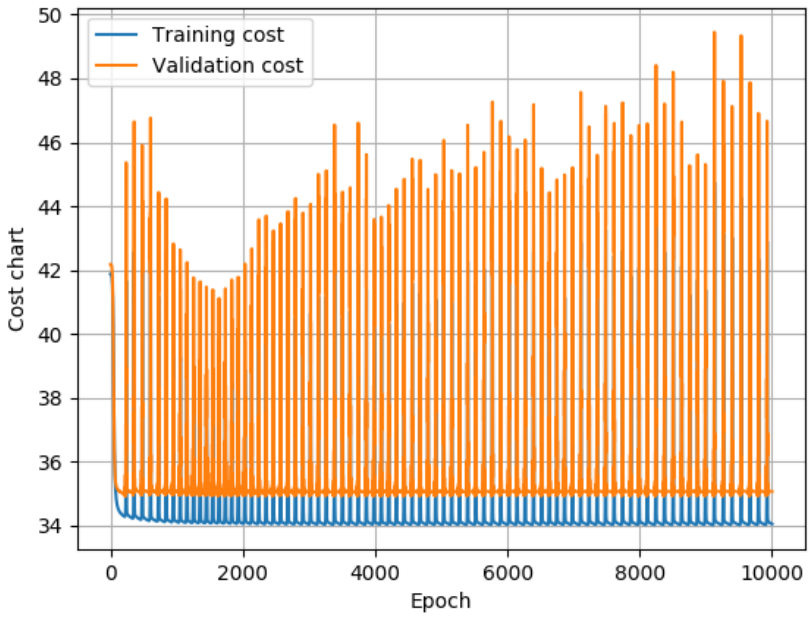

```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[ 9.92547276]]

X=[[ -2.28749635e-03],
 [  6.04665145e+00]]
real value:  [-0.00457494]
prediction: [[ 0.11506481]]

X=[[ 2.93511782],
 [-1.8467719 ]]
real value:  [-5.26989202]
prediction: [[-5.19914678]]

X=[[-3.72520423],
 [-6.91121454]]
real value:  [ 6.78000488]
prediction: [[ 6.96462379]]

Regression metrics:
MSE: 9.68152041729e-05
R2:  0.999611478774
Pearson: 1.000, p-value: 0.000
Spearman: 0.999, p-value: 0.000
```

Dobry wynik, jednak nieznacznie gorszy od Modelu 4.

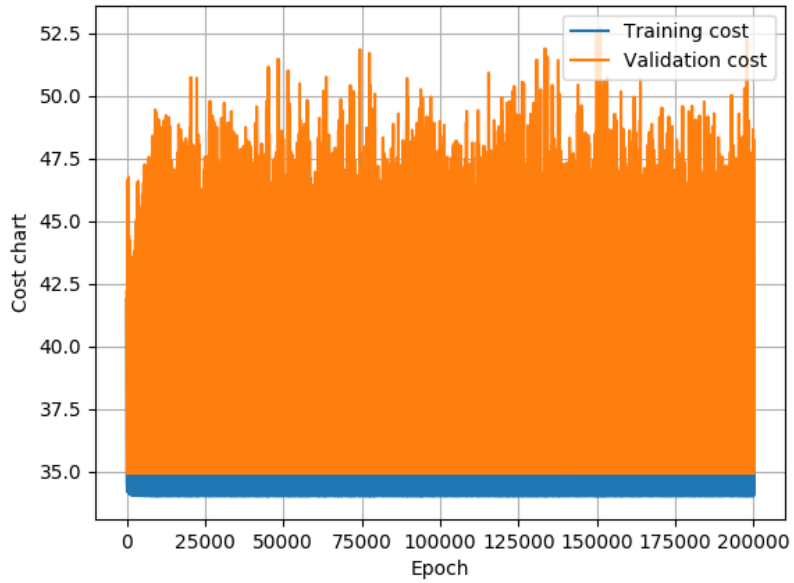
2.3.9. Architektura [2, 2, 1], funkcja aktywacji: ReLU

```
number_of_iterations = 10000
learning_rate = 0.01
```

Po 10.000 iteracji:



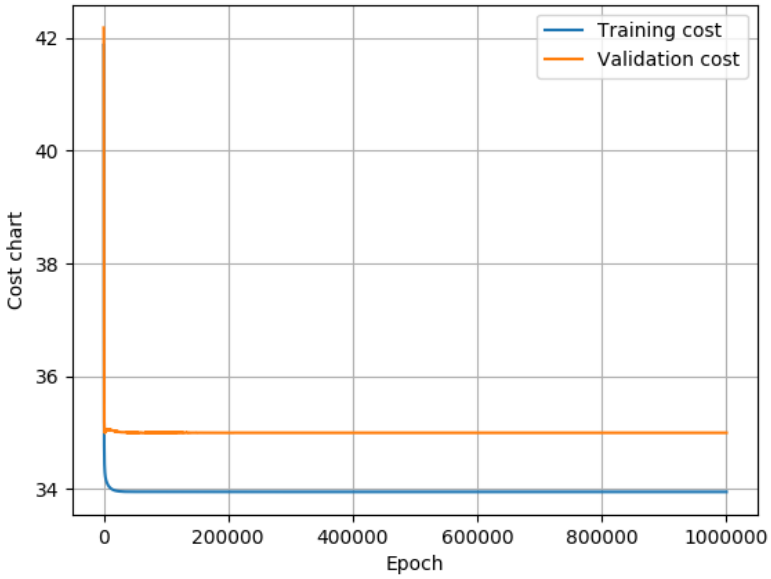
Po 200.000 iteracji:



Próbujemy jeszcze zmniejszyć współczynnik uczenia:

```
learning_rate = 0.001
```

Nawet zwiększenie liczby iteracji do 1.000.000 nie daje zadnych ciekawych rezultatów:

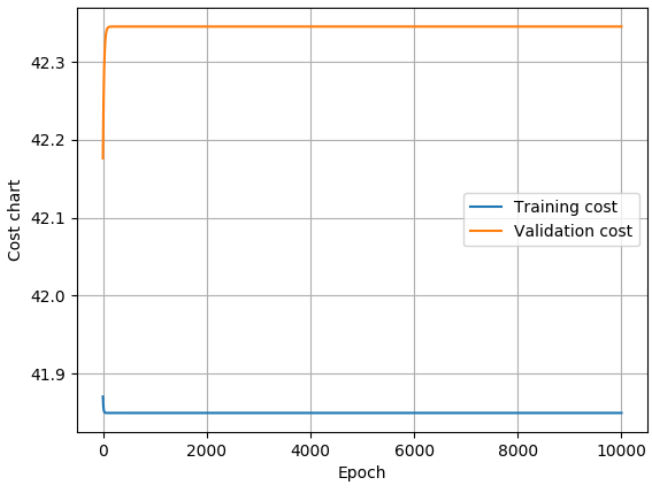


2.3.10. Architektura [2, 2, 3, 3, 5, 5, 3, 3, 2, 2, 1], funkcja aktywacji: ReLU

```
number_of_iterations = 10000
learning_rate = 0.05
```

Wynik taki sam jak w Modelu 6 (ta sama architektura, ale funkcja Leaky ReLU).

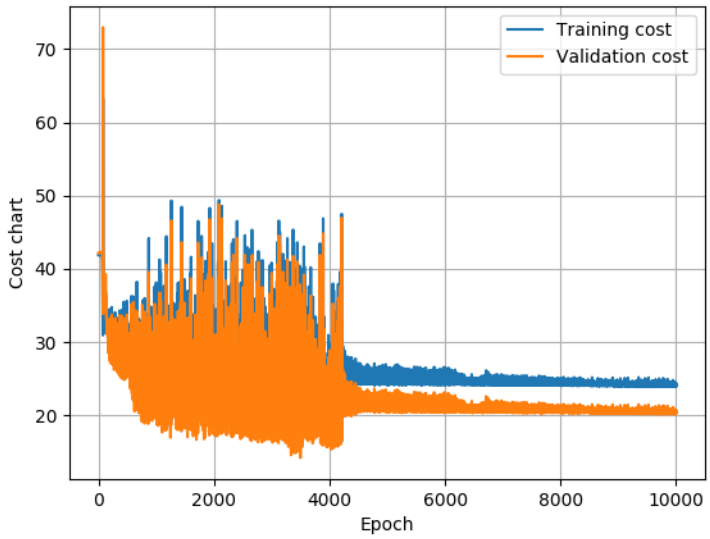
Po 10.000 iteracjach:



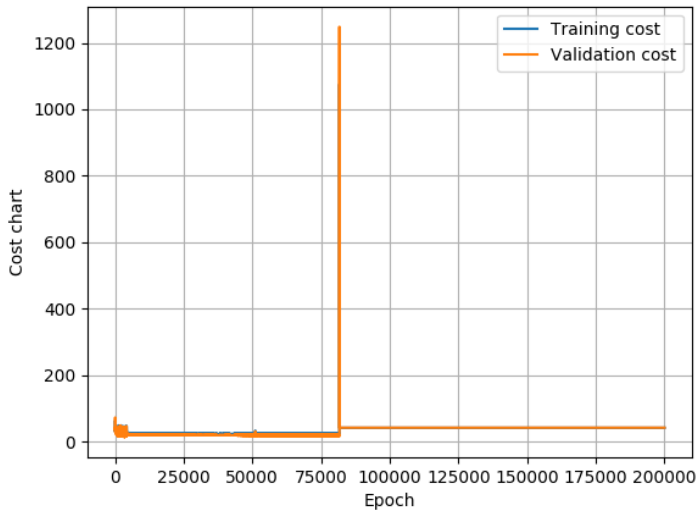
2.3.11. Architektura [2, 20, 20, 1], funkcja aktywacji: ReLU

```
number_of_iterations = 10000
learning_rate = 0.05
```

Po 10.000 iteracjach:



Kontynuujemy do 200.000 iteracji:



Po 80000 iteracjach model przeskakuje do innego rozwiązania ozyskując gorszy wynik:

```
Iteration: 81660   Training cost: 23.27897 Testing cost: 18.30332
Iteration: 81665   Training cost: 24.43265 Testing cost: 19.72811
Iteration: 81670   Training cost: 22.93837 Testing cost: 17.94798
Iteration: 81675   Training cost: 23.25714 Testing cost: 18.28023
Iteration: 81680   Training cost: 26.17694 Testing cost: 25.55619
```

```
Iteration: 81685   Training cost: 42.54666 Testing cost: 41.94676
Iteration: 81690   Training cost: 42.26707 Testing cost: 41.91533
Iteration: 81695   Training cost: 42.09968 Testing cost: 41.93991
Iteration: 81700   Training cost: 41.99945 Testing cost: 41.98821
Iteration: 81705   Training cost: 41.93943 Testing cost: 42.04310
```

Jednak nawet przed przeskokiem wynik nie był zadowalający.

Finalnie model (ten po 200.000 iteracjach) uzyskuje poniższe wyniki:

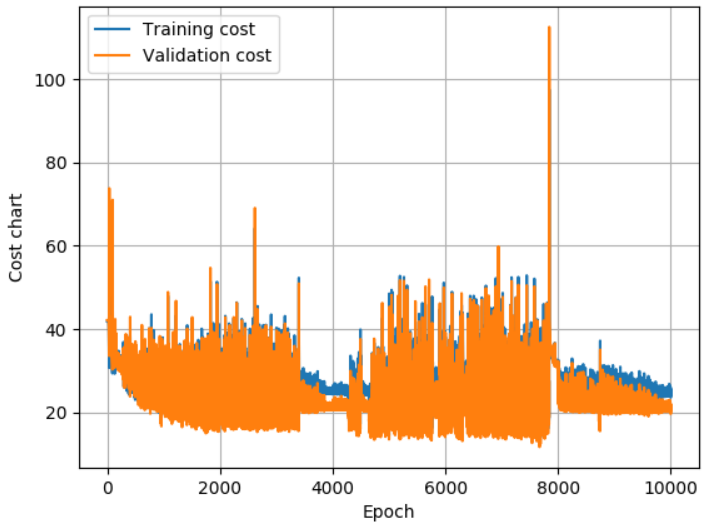
```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[-0.1449533]]

X=[[ -2.28749635e-03],
 [  6.04665145e+00]]
real value:  [-0.00457494]
prediction: [[-0.1449533]]

X=[[ 2.93511782],
 [-1.8467719 ]]
real value:  [-5.26989202]
prediction: [[-0.1449533]]

X=[[-3.72520423],
 [-6.91121454]]
real value:  [ 6.78000488]
prediction: [[-0.1449533]]

Regression metrics:
MSE: 0.252055861901
R2:  0.0100566889734
```

Czyli nieciekawe. Dodatkowo wbudowany w pakiet scipy funkcje do liczenia korelacji wyrzuciły wyjątki.

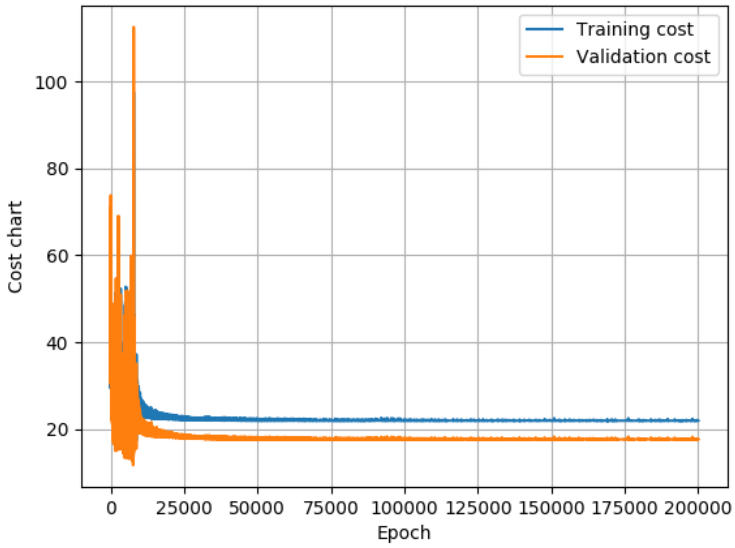
2.3.12. Architektura [2, 100, 100, 1], funkcja aktywacji: ReLU

```
number_of_iterations = 10000
learning_rate = 0.05
```

Po 10.000 iteracjach:



Nie zaciekawie, ale przedłużamy do 200.000 iteracji:




```
Sanity check:
X=[[  8.34044009],
 [ 14.40648987]]
real value:  [ 9.95270895]
prediction: [[ 9.17587057]]

X=[[ -2.28749635e-03],
 [  6.04665145e+00]]
real value:  [-0.00457494]
prediction: [[-0.19128286]]

X=[[ 2.93511782],
 [-1.8467719 ]]
```

```
real value:  [-5.26989202]
prediction: [[-5.49608352]]

X=[[-3.72520423],
 [-6.91121454]]
real value:  [ 6.78000488]
prediction: [[-0.19128286]]

Regression metrics:
MSE: 0.105735987548
R2:  0.570828202237
Pearson: 0.762, p-value: 0.000
Spearman: 0.738, p-value: 0.000
```

Niestety nawet najbardziej obiecująca z wybranych architektur dla funkcji ReLU nie dała dobrego modelu.

2.4 Wybór najlepszego modelu

Dodawanie warstw ukrytych nie dawało dobrych modeli. Lepsze okazywało się dodawanie neuronów dla dwóch warstw ukrytych. Zdecydowanie najlepszą spośród badanych architektur to [2, 100, 100, 2]. Najlepsze wyniki uzyskaliśmy dla funkcji aktywacji Sigmoid (Model 4). Równie dobry wynik uzyskaliśmy dla tej samej architektury, ale z funkcją Leaky ReLU (Model 8). Niestety zmiana funkcji aktywacji na ReLU dała złe wyniki nawet w architekturze [2, 100, 100, 2], która sprawdziła się z funkcjami Sigmoid i Leaky ReLU.

Poniżej podsumowanie najlepszych modeli:

Model 3:
Architektura: [2, 20, 20, 1], funkcja aktywacji: Sigmoid
MSE: 0.000149555575082
R2: 0.99939999844
Pearson: 1.000, p-value: 0.000
Spearman: 0.999, p-value: 0.000

Uzyskany po 200.000 iteracjach przy learning_rate = 0.05

Model 4:
Architektura: [2, 100, 100, 1], funkcja aktywacji: Sigmoid
MSE: 5.84025867309e-05
R2: 0.999765782618
Pearson: 1.000, p-value: 0.000
Spearman: 0.999, p-value: 0.000

Uzyskany po 200.000 iteracjach przy learning_rate = 0.05

Model 7:
Architektura [2, 20, 20, 1], funkcja aktywacji: Leaky ReLU
MSE: 0.0024805885721
R2: 0.990221575021

Pearson: 0.996, p-value: 0.000
Spearman: 0.992, p-value: 0.000

Uzyskany po 200.000 iteracjach przy learning_rate = 0.001

Model 8:
Architektura [2, 100, 100, 1], funkcja aktywacji: Leaky ReLU
MSE: 9.68152041729e-05
R2: 0.999611478774
Pearson: 1.000, p-value: 0.000
Spearman: 0.999, p-value: 0.000

Uzyskany po 200.000 iteracjach przy learning_rate = 0.01

Uzyskane wyniki są zbliżone (nieco większy błąd średniokwadratowy w Modely 7 i nieco mniejsza korelacja). Należy zauważyć, że dla Leaky RELU (Model 7 i Model 8) dobry wynik osiągnęliśmy dopiero po zmniejszeniu współczynnika uczenia do odpowiednio 0.001 i 0.01.

Optymalna struktura dla problemu to Model 4, czyli [2, 100, 100, 1] z funkcją aktywacji: Sigmoid.

2.5 Szczegóły implementacyjne

Sieć neuronowa została napisana samodzielnie z wykorzystaniem jedynie biblioteki NumPy do operacji na macierzach. Proces uczenia polega na aktualizacji współczynników w każdej iteracji na bazie gradientów wyliczonych za pomocą tzw. back-propagation, który został opisany na diagramie na stronie 5 w podrozdziale 1.4. Opis algorytmu.

Do fragmentów dających się policzyć ręcznie (np. forward propagation, czy funkcje aktywacji) dopisaliśmy testy jednostkowe. Wykorzystany został framework unittest.

Np. dla funkcji sigmoidalnej:

```
def sigmoid(Z):
  return 1/(1+np.exp(-Z))

def sigmoid_backward(dA, Z):
  return dA * derivative(Z)

def derivative(Z):
  S = sigmoid(Z)
  return S * (1-S)
```

mamy testy postaci (jeden dla funkcji sigmoid, drugi dla backward, który korzysta z pochodnej):

```
  def test_sigmoid(self):
    X = np.array(
         [0, 0.19721045, 0.17785665, -0.23369414, 0.55528333, 2.07653441,
         -1.23360985, -0.84550586, -1.45364079, -0.61059541, 1.42725904])
    expected = np.array(
         [0.5, 0.549143441907707, 0.544347320897941, 0.441840911014308,
         0.635360493036452, 0.888601439999118, 0.225550242127031,
         0.300376454902037, 0.189441877204342, 0.35192338881577,
         0.806473883161429])
```

```
    Y = sigmoid(X)
    result = np.all(np.isclose(Y, expected))
    self.assertTrue(result)

  def test_backward(self):
    dA = np.array(
         [0, 3.54531542, -4.81741295, -2.73031028, -16.23282883, 4.96819628,
         3.85990703, -16.42697689, -2.91987025, -3.20573417, 8.31951293])
    Z = np.array(
         [-1.25541855, 1.15025199, 0.64849487, 0.82883564, 0.11937011, 0.2328452,
         2.40720121, -0.14718964, 0.92134979, -1.48488645, -0.630742])
    expected = np.array(
         [0., 0.64748164, -1.08610566, -0.57757362, -4.0437849, 1.22536502,
         0.29257434, -4.08458135, -0.59459819, -0.48271916, 1.88599722])
    Y = sigmoid_backward(dA, Z)
    result = np.all(np.isclose(Y, expected))
    self.assertTrue(result)
```

Inny przykładowy test, tym razem dla forward propagation:

```
    X = np.array([[-1.2], [3.5]])
    W = {}
    b = {}
    L = 3
    W[1] = np.array([[0.1, 1.7], [8.2, -0.4], [-5.5, 1.7]])
    b[1] = np.array([[-3.2], [0.2], [-1.9]])
    W[2] = np.array([[0.5, 1.7, -1.1], [-0.4, -0.3, 1.1], [7.1, 3.3, -4.7]])
    b[2] = np.array([[1.0], [3.9], [-3.0]])
    W[3] = np.array([[-2.2, 2.7, 0.1]])
    b[3] = np.array([[-0.4]])
    A, _ = forward(X, L, W, b, linear)
    result = np.all(np.isclose(A[L], np.array([[102.7507]])))
    self.assertTrue(result)
```

Implementacja metryk:

Współczynnik determinacji oraz błąd średniokwadratowy zostały zaimplementowane zgodnie ze wzorami natomiast korelacje Spearmana i Persona zostały wyliczone przy uzyciu biblioteki SciPy:

```
import numpy as np
from scipy.stats import pearsonr, spearmanr
from data_gen import generate_data_sets, generate_data_helper
from forward import forward

def print_metrics(layers, W, b, activation_fun):
  L = len(layers) - 1  # number of layers - input layer doesn't count
  train_data_min = -20
  train_data_max = 20
  train_data_step = 1
  test_data_min = -20
  test_data_max = 20
  test_data_step = 1
  trX, trY, tsX, tsY, validation_data =
generate_data_sets(generate_data_helper(train_data_min, train_data_max,
train_data_step, test_data_min, test_data_max, test_data_step))
  tsA, _ = forward(tsX, L, W, b, activation_fun['forward'])
  print("Regression metrics:")
  print("MSE:", np.power(tsA[L] - tsY, 2).mean() / np.shape(tsA[L])[1])
```

```
  SSreg = np.sum(np.power(tsA[L] - tsY.mean(), 2))
  SSres = np.sum(np.power(tsA[L] - tsY, 2))
  SStot = SSres + SSreg
  R2 = 1 - SSres / SStot
  print("R2: ", R2)
  pearson, pearson_pvalue = pearsonr(tsA[L].reshape(tsA[L].shape[1]),
tsY.reshape(tsY.shape[1]))
  spearman, spearman_pvalue = spearmanr(tsA[L].reshape(tsA[L].shape[1]),
tsY.reshape(tsY.shape[1]))
  print("Pearson: {:.3f}, p-value: {:.3f}".format(pearson, pearson_pvalue))
  print("Spearman: {:.3f}, p-value: {:.3f}".format(spearman, spearman_pvalue))
```

2.6. Zbiór trenujący i testowy

Dane zostały wygenerowane na bazie „kratki” pomiędzy punktami (-20, -20), (-20, 20), (20, -20), (20, 20). Odstęp pomiędzy punktami wynosi 1. Tak wygenerowana kratka została losowo podzielona na 3 podzbiory, których wielkości w stosunku do zbioru wyjściowego wynoszą mniej więcej: 80%, 10%, 10% i są to odpowiednio zbiór trenujący, testowy, walidacyjny.
Finalnie skorzystaliśmy tylko z dwóch pierwszych, czyli trenującego (~80%) i testowego (~10%). Przybliżenie wynika ze sposobu podziału – wygenerowane wartości są losowo przydzielane do zbiorów train, test, validation z rozkładem 0.8, 0.1, 0.1.

2.7 Wnioski

Sieć neuronowa okazała się dobrym regresorem dla badanej funkcji, aczkolwiek dużo zależy od architektury (liczba warstw, neuronów oraz funkcji aktywacji), a także zbioru posiadanych danych trenujących. W naszym przypadku danych olabelkowanych (uczących) było ponad 1200 i to wystarczyło dla analizowanej funkcji.

NumPy okazała się dobrą biblioteką, dzięki której operacje na macierzach wykonują się szybko, co osiąga się dzięki Streaming SIMD Extentions, a także dość wygodnie z punktu widzenia programisty dzięki takim ułatwieniom jak broadcasting (dodawanie skalara do macierzy powoduje rozszerzenie skalara do wektora), czy dzięki wielu wbudowanym funkcjom wykonujących operacje na całych macierzach (jak np. mnożenie: numpy.dot), czy na każdych elementach (tzw. element-wise, np. podnoszenie elementów macierzy do potęgi: numpy.power).