**Podstawy Sztucznej Inteligencji – Laboratorium nr 3**

Wykonał: Kamil Wieniecki

Temat ćwiczenia: Budowa i działanie sieci wielowarstwowej typu feedforward.

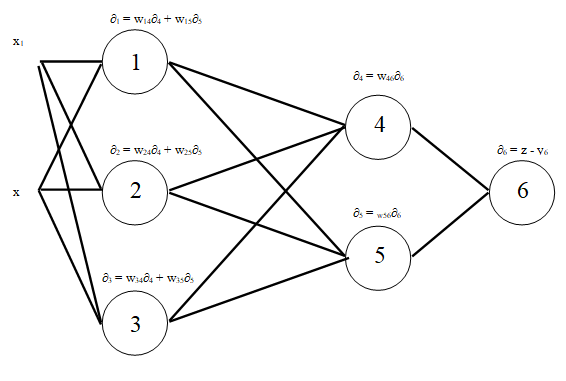
1. Cel ćwiczenia

Celem ćwiczenia jest poznanie budowy i działania wielowarstwowych sieci neuronowych poprzez uczenie kształtu wykresu funkcji matematycznej z użyciem wstecznej propagacji błędu.

1. Realizacja ćwiczenia

Wybrany przeze mnie język programowania do wykonania laboratorium to **Python.** Implementacja wykorzystuje bibliotekę open-source o nazwie **Keras**, która służy do deep neural learningu.

Biblioteka ta wykorzystuje algorytm wstecznej propagacji błedów (backpropagation), która przedstawia się następująco (przykład trójwarstwowej sieci neuronowej z dwoma wejściami i jednym wyjściem):



,gdzie:

δi – wartość błędu i-tego neuronu,

wab – waga neuronu a w neuronie b,

z – wartość oczekiwana,

yi – wartość wyjścia neuronu,

Testowane przeze mnie struktury sieci to:

- 30 - 1

- 30 – 10 - 1

- 30 – 30 – 30 - 1

Oraz kolejno współczynniki uczenia dla tych prób:

- 0.01

- 0.1

- 0.4

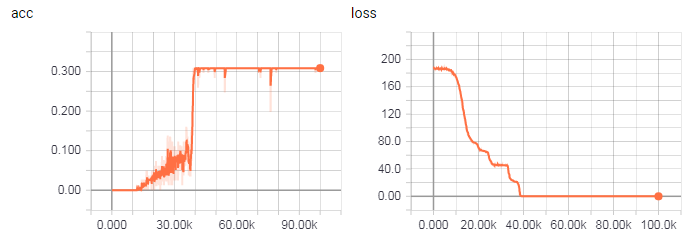
Każda próba uczenia opierała się na XXXX epokach, dla XXXX punktów wygenerowanych losowo dla tychże testów. Walidacja opierała się na siatce punktów ∂x = 0.5 oraz ∂y = 0.5. Biblioteka pozwala na określenie wielkości batch\_size, która to odpowiedzialna jest za aktualizacje wag po ilości próbek podanych jako ten właśnie argument. Implementacja wykorzystuje średni błąd kwadratowy, czyli różnicę pomiędzy estymatorem (wartością policzoną) i wartością estymowaną (wartością szacunkową).

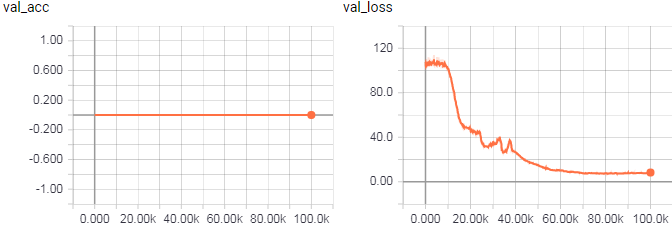
Każdy model został zapisany do pliku .h5, który może zostać w łatwy sposób odczytany przy pomocy metody bibliotecznej metody model\_load. Wyniki zostały zapisane w katalogu z logami, gdzie korzystając z narzędzia **TensorBoard** możemy stworzyć na ich podstawie wykresy.

1. Wyniki

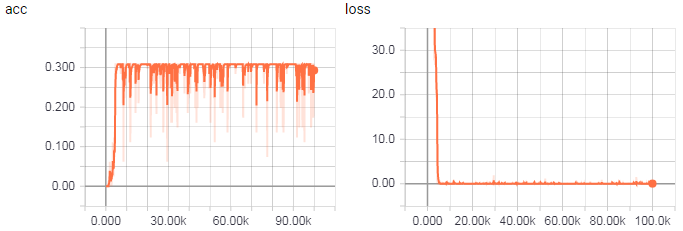
Wyniki z uczenia prezentują się następująco:

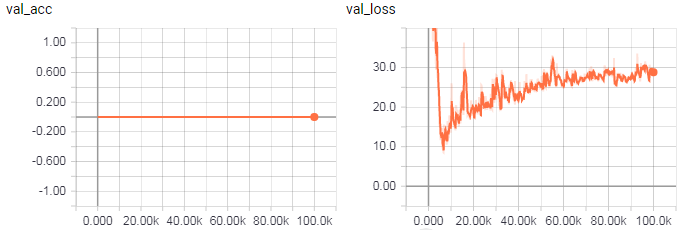
Dla struktury 30-1, learning rate = 0.01



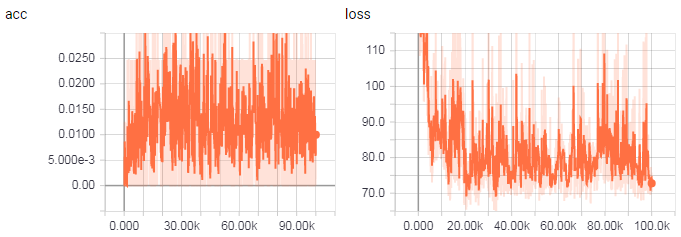


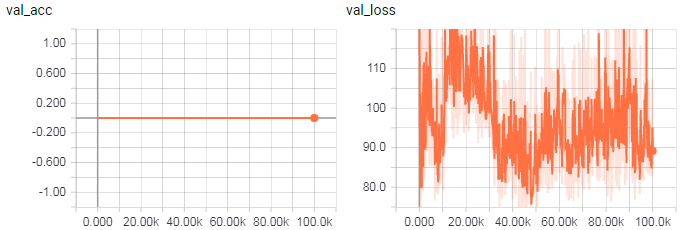
Dla struktury 30-1, learning rate = 0.1



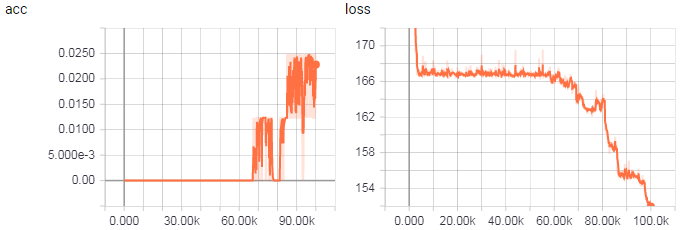
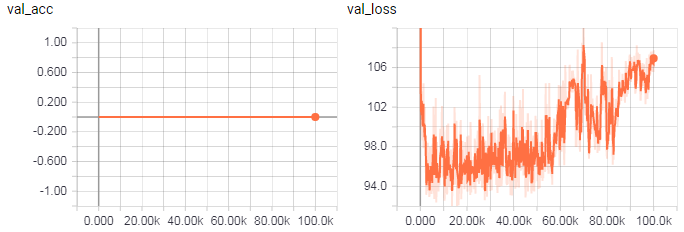


Dla struktury 30-1, learning rate = 0.4

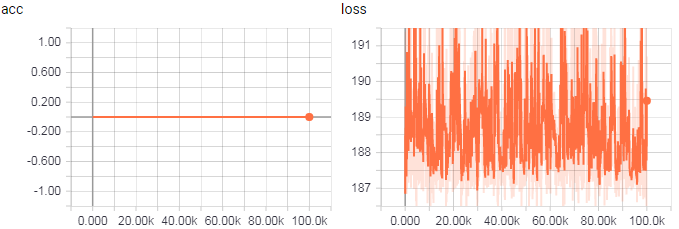


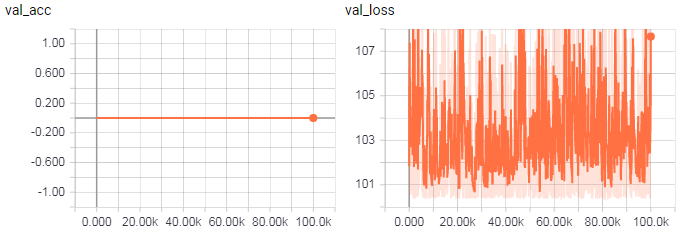


Dla struktury 30-10-1, learning rate = 0.01

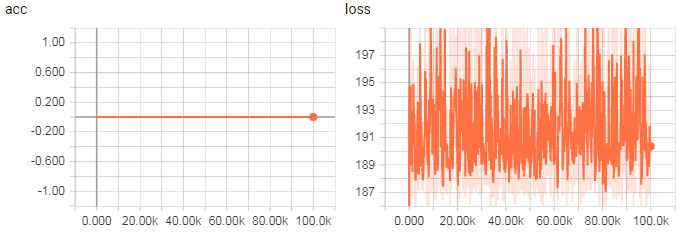
 

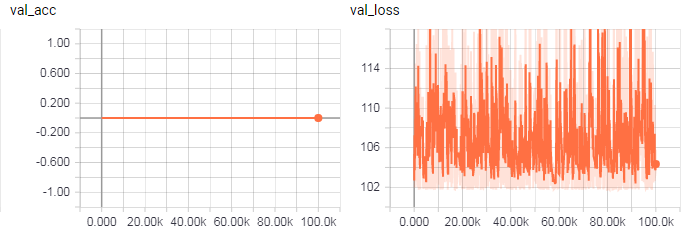
Dla struktury 30-10-1, learning rate = 0.1



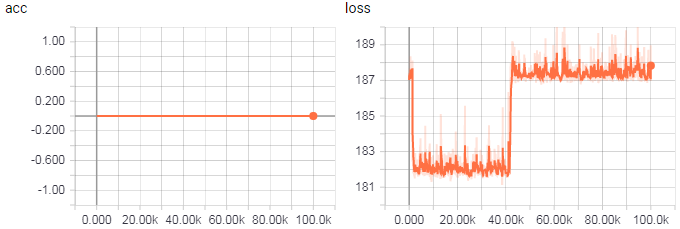


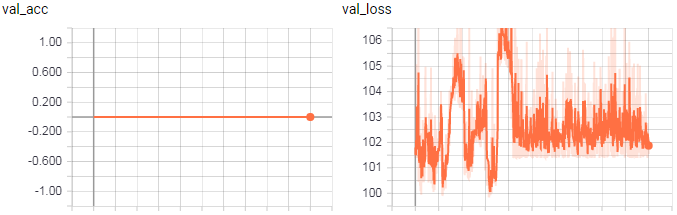
Dla struktury 30-10-1, learning rate = 0.4



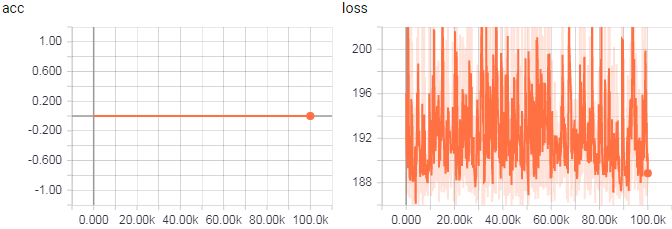


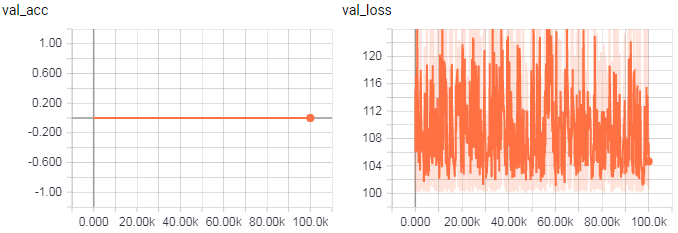
Dla struktury 30-30-30-1, learning rate = 0.01



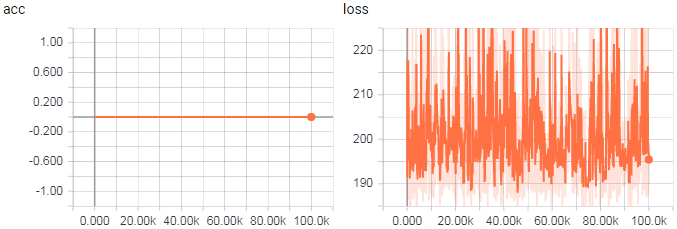


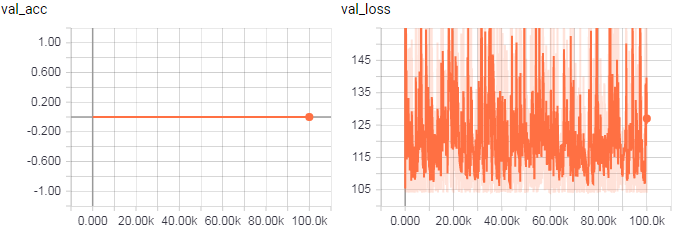
Dla struktury 30-30-30-1, learning rate = 0.1





Dla struktury 30-30-30-1, learning rate = 0.4





1. Analiza wyników

Mniej skomplikowane struktury perceptronów (w tym przypadku 30-1) osiągnęły znacznie lepszy wynik niż pozostałe. Skomplikowane struktury nie zbliżyły się dokładnością do zadanej funkcji w żadnym stopniu (wyjątkiem jest struktura 30-10-1 z learning rate = 0.01, dla której wyniki zaczęły być nieco lepsze, aczkolwiek dokładność na poziomie 2,5% jest zdecydowanie za niska). Osiągniecie wyniku około 30% dokładności dla najlepszej struktury nie jest zadowalającym wynikiem. Z bardziej rozbudowanych struktur tak naprawdę nie można odczytać żadnych istotnych obserwacji, ponieważ jak widać po wykresach – przy 100 000 epok nauka nie zakończyła się powodzeniem.

1. Wnioski

Przeciętny czas nauki jednej sieci to około 2,5h.

Niektóre struktury nie nadawały się całkowicie do nauki, gdyż implementacja biblioteki Keras nie radziła sobie z uczeniem na podstawie wprowadzonych danych. Najlepiej wypadła struktura perceptronów [ 30 – 1 ], w której to najlepiej widać. Chociaż współczynnik dokładności na poziomie około 30% nie jest najlepszym wynikiem. Błąd MSE dla tej struktury osiągnął bardzo małą wartość bliską 0. Funkcja Rastrigin jest również trudną do wyuczenia funkcją, ponieważ posiada bardzo mnóstwo minimów i maksimów lokalnych co znacznie utrudnia skuteczną naukę. Większa wartość współczynnika nauki wcale nie wpływa lepiej na wyniki. Na podstawie wykresów, można wręcz rzecz, że funkcja Rastrigin wymaga niewielkiego współczynnika nauki, ale bardzo długiego uczenia. Delikatna modyfikacja wag znacznie lepiej wpływa na naukę, co jest podkreślone wykresami dla struktury 30-1 learning rate = 0.01. Większa dynamika zmiany learning rate (0.4) spowodowała ogromne skoki na wykresie i bardzo niską dokładność nauki. Również błąd MSE był zdecydowanie za duży. Błąd MSE najszybciej osiągnął wartość bliską 0 dla struktury 30-1 z learning rate = 0.1.

1. Listing kodu

**Rastrigin.py**

"Import matematycznej biblioteki Python"

import math

def get\_equation(x, y):

"Równanie"

return 20 + x \* x + y \* y - 10 \* (math.cos(math.pi \* 2 \* x) + math.cos(math.pi \* 2 \* y))

**TestData.py**

from random import uniform

from Rastrigin import get\_equation

random\_input = []

random\_output = []

test\_input = []

test\_output = []

def my\_range(start, end, step):

"Funkcja do manipulacji kroku dla petli for"

while start <= end:

yield start

start += step

def fill\_random\_test\_input(num):

"Wypełnienie losowymi wartościami"

for i in range(0, num):

first = uniform(-2, 2)

second = uniform(-2, 2)

random\_input.append([first, second])

random\_output.append(get\_equation(first, second))

def fill\_test\_input(num):

"Wypelnienie testowych wartosci"

for i in my\_range(-2, 2, num):

for j in my\_range(-2, 2, num):

test\_input.append([i, j])

test\_output.append(get\_equation(i, j))

**Main.py**

from keras.models import Sequential

from keras.layers.core import Dense

from keras.callbacks import TensorBoard

from keras import optimizers

from prettytable import PrettyTable

from TestData import \*

## Dane wejściowe

fill\_test\_input(0.5)

fill\_random\_test\_input(1000)

LAYERS = [30, 30, 30, 1]

LEARNING\_RATE = 0.4

BATCH\_SIZE = 20

ERAS = 100000

DECAY = 0

MODEL = Sequential()

LOG\_DIR = "./logs-lr" + str(LEARNING\_RATE) + "-lay"

for lay in LAYERS:

LOG\_DIR += "-" + str(lay)

TENSORBOARD = TensorBoard(log\_dir=LOG\_DIR,

histogram\_freq=5,

batch\_size=100,

write\_graph=True,

write\_grads=False,

write\_images=True,

embeddings\_freq=0,

embeddings\_layer\_names=None,

embeddings\_metadata=None

)

for i in range(0, len(LAYERS)):

if i == 0:

MODEL.add(Dense(LAYERS[i], input\_dim=2, activation="sigmoid"))

elif i == len(LAYERS) - 1:

MODEL.add(Dense(LAYERS[i], activation="linear"))

else:

MODEL.add(Dense(LAYERS[i], activation="sigmoid"))

ADAM = optimizers.Adam(lr=LEARNING\_RATE, beta\_1=0.9, beta\_2=0.999, epsilon=1e-08, decay=DECAY)

MODEL.compile(loss='mean\_squared\_error', optimizer=ADAM, metrics=['accuracy'])

MODEL.fit( test\_input,

test\_output,

epochs=ERAS,

batch\_size=BATCH\_SIZE,

validation\_data=(random\_input, random\_output),

callbacks=[TENSORBOARD]

)

RESULTS = MODEL.evaluate(random\_input, random\_output)

print("\n%s: %.2f%%" % (MODEL.metrics\_names[1], RESULTS[1]\*100))

print(MODEL.summary())

print("Learning rate =", LEARNING\_RATE)

YHAT = MODEL.predict(random\_input, verbose=0)

PRETTYTABLE = PrettyTable()

PRETTYTABLE.field\_names = ['x1', 'x2', 'PREDICTED', 'EXPECTED']

for i in range(0, len(YHAT)):

PRETTYTABLE.add\_row([random\_input[i][0], random\_input[i][1], YHAT[i][0], random\_output[i]])

print(PRETTYTABLE)

MODEL.save('model-sieci-' + str(LAYERS) +'-lr-' + str(LEARNING\_RATE) + '-decay-' + str(DECAY) + '.h5')

1. Bibliografia

<https://en.wikipedia.org/wiki/Rastrigin_function>

https://keras.io/