Podstawy Sztucznej Inteligencji – Laboratorium nr 3

Wykonał: Paweł Nowak

Temat ćwiczenia: Budowa i działanie sieci wielowarstwowej typu feedforward.

1. Cel ćwiczenia

Celem ćwiczenia było poznanie budowy i działania wielowarstwowych sieci neuronowych poprzez uczenie z użyciem algorytmu propagacji wstecznej błędu rozpoznawania konkretnych liter alfabetu.

2. Realizacja ćwiczenia

Do wykonania zadania użyto biblioteki PyBrain w Pythonie.
W pierwszej kolejności utworzono plik zawierający dane wejściowe, w tym wypadku wielkie litery A, B, C, D, E, F, G, H, I, J, K, U, M, L, O, P, R, T, W, S. Na każdą literę składają się 2 elementy: macierz 5x7 z liczb 1 lub -1 przedstawiająca wygląd litery oraz tablica.
PyBrain zawiera paczkę datesets, w której znajduje się klasa SupervisedDataSet odpowiedzialna za przechwywanie danych.
Każda z liter dodawana jest do zbioru danych wejsciowych za pomocą addSample().
Przykładowo:

```
daneWejsciowe.addSample((
                                -1, 1, 1, 1, -1,
                                1, -1, -1, -1, 1,
                                1, -1, -1, -1, 1,
                                1, 1, 1, 1, 1,
                                1, -1, -1, -1, 1,
                                1, -1, -1, -1, 1,
                                1, -1, -1, -1, 1
                            ),
                                (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                            0, 0))
```

Następnie za pomocą metody FeedForwardNetwork utworzono wielowarstwową sieć neuronową składającą się z warstwy wejsciowej, ukrytej oraz wyjściowej.

Dodano również bias stworzony wcześniej metodą BiasUnit().

Można dodać kilka modułów wejściowych oraz wyjściowych. Sieć musi wiedzieć który z nich jest wejściem a który wyjściem aby poprawnie przesyłać wejście oraz propagować wstecznie błąd. Lecz aby ich używać musimy dodać je do sieci, robimy to używając addInputModule(), addModule(), addOutputModule().

Zatem należy zdeterminować w jaki sposób mają być one połączone. Robimy to za pomocą klasy FullConnection. Łączymy ze sobą warstwy w następujący sposób:

```
bias_ukryty = FullConnection(bias, ukryty)
wejscie_ukryty = FullConnection(wejscie, ukryty)
ukryty_wyjscie = FullConnection(ukryty, wyjscie)
```

Następnie dodajemy połączenia do sieci:

```
        siec.addConnection(bias_wejscie)
        siec.addConnection(bias_ukryty)
siec.addConnection(bias_wyjscie)
```

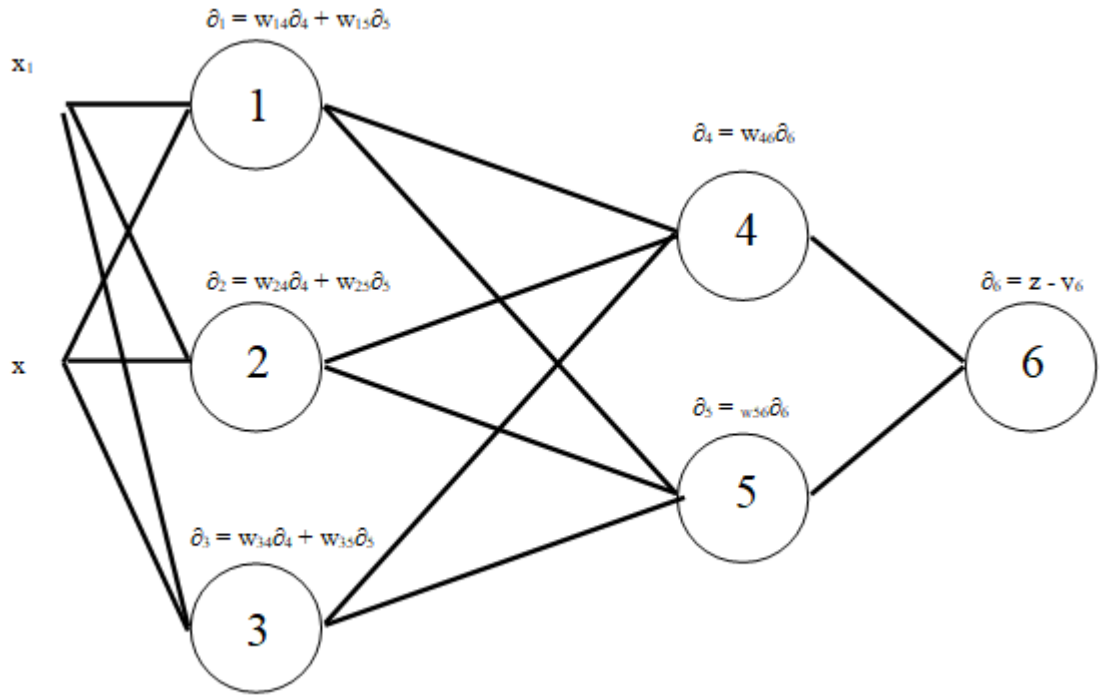
Wszystkie elementy zostały dodane, zatem ostatnim co należy zrobić to użyć metody sortModules(), która spowoduje wewnętrzną inicjalizację, która jest niezbędna do poprawnego działania sieci.

Sieć jest trenowana za pomocą algorytmu propagacji wstecznej w przypadku PyBrain'a importowany jest BackpropTrainer który przyjmuje sieć, dane wejsciowe współczynnik uczenia. Nastepnie przy użyciu trainEpochs trenujemy sieć.

Następnie przy pomocy activate() możemy obliczyć output.

Dla każdej litery wyswietlany jest output każdej możliwej opcji (przykładowy output na końcu)

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



,gdzie:

δ_i – wartość błędu i-tego neuronu,

w_{ab} – waga neuronu a w neuronie b,

z – wartość oczekiwana,

y_i – 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

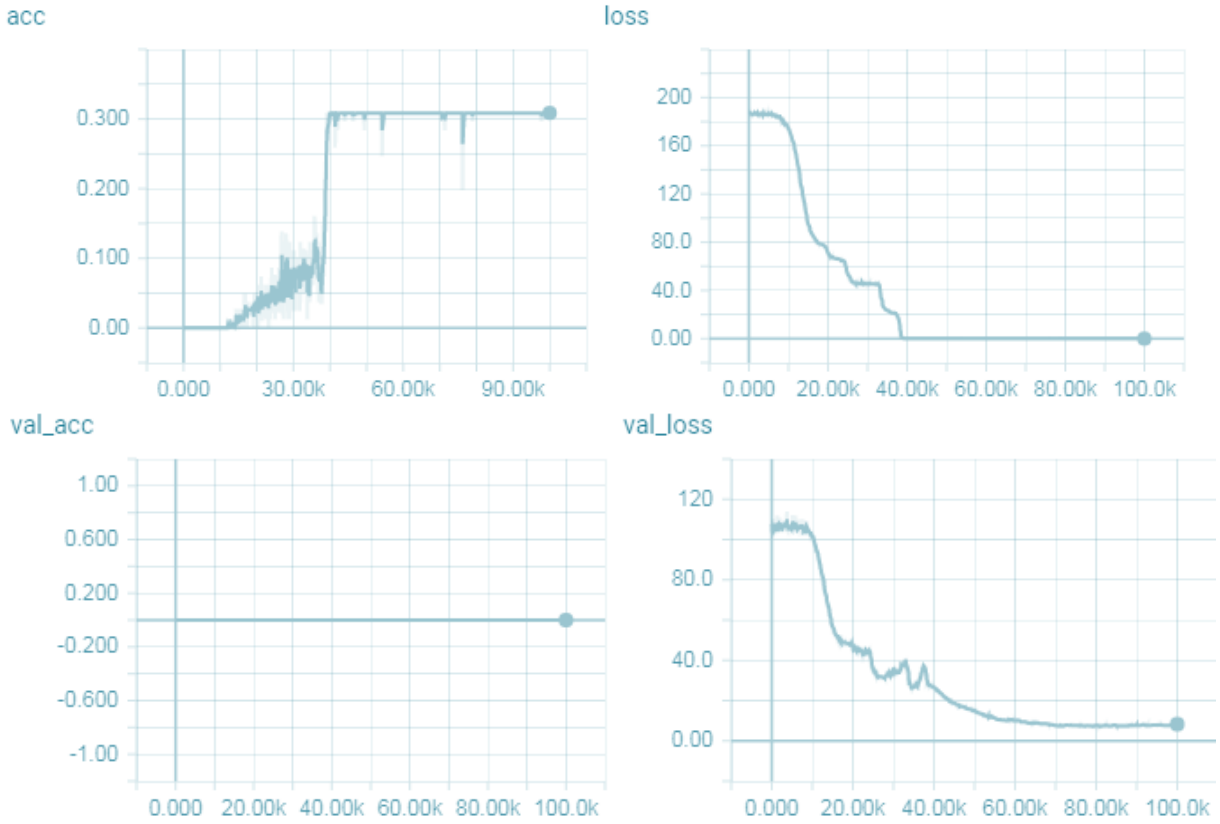
pozwala na określenie wielkości batch_size, która to odpowiedzialna jest za aktualizacje wag po ilości próbek podanych jako ten właśnie argument. Implementacja wykorzystuje średni błąd kwadratowy, czyli różnicę pomiędzy estymatorem (wartością policzoną) i wartością estymowaną (wartością szacunkową).

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.

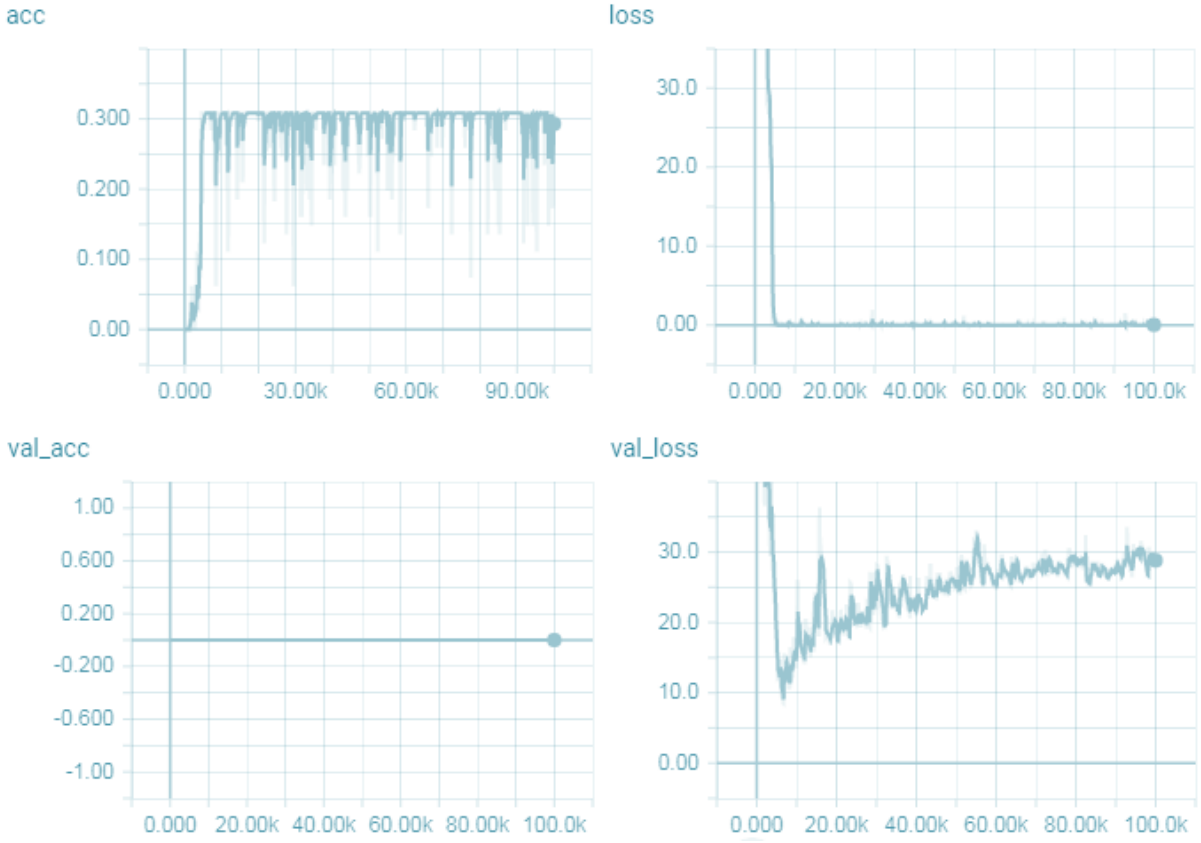
3. Wyniki

Wyniki z uczenia prezentują się następująco:

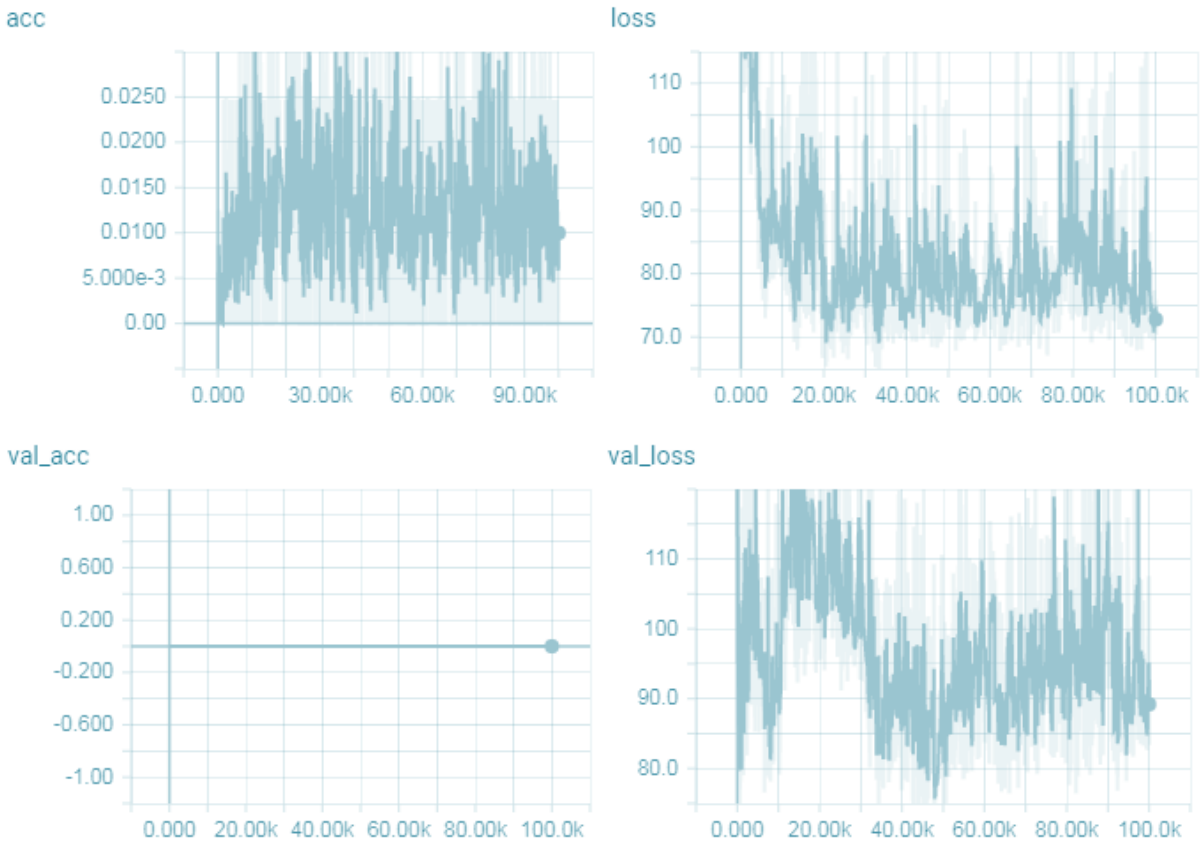
Dla struktury 30-1, learning rate = 0.01



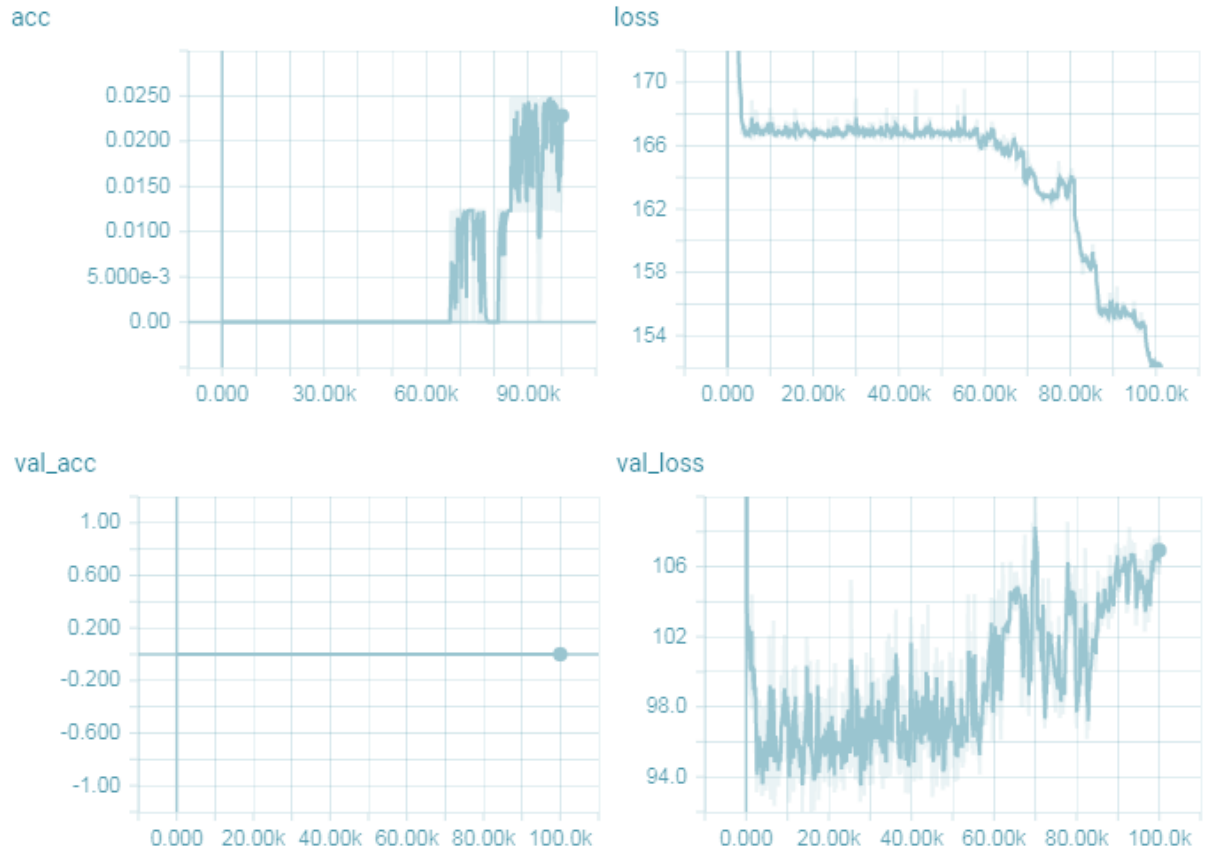
Dla struktury 30-1, learning rate = 0.1



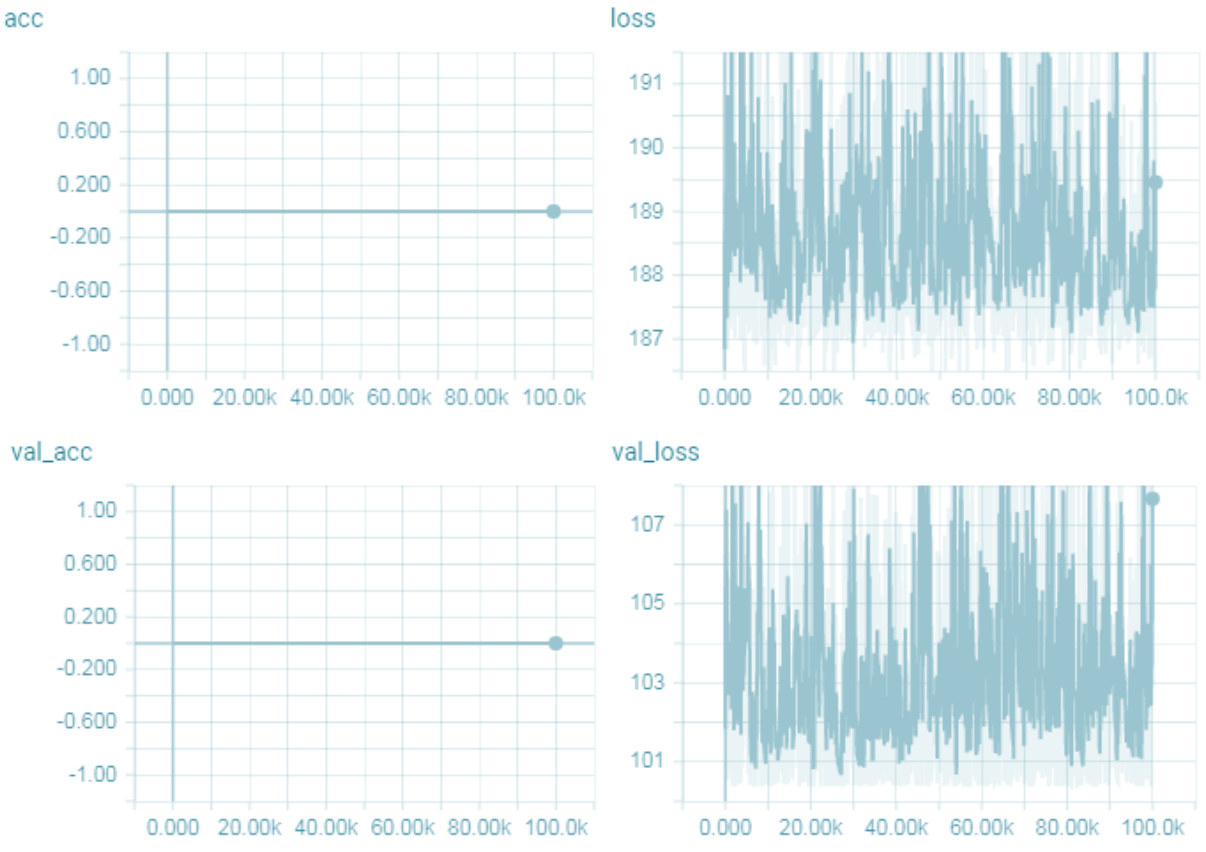
Dla struktury 30-1, learning rate = 0.4



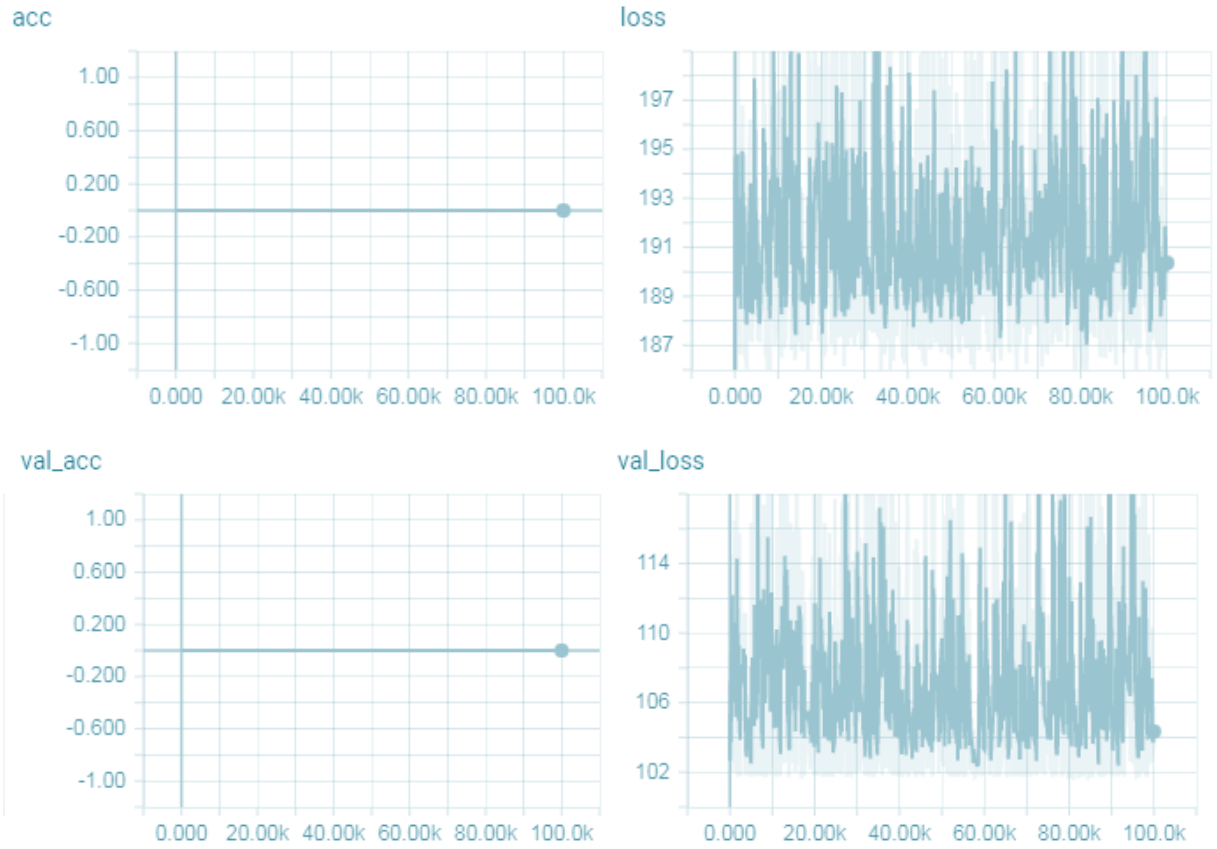
Dla struktury 30-10-1, learning rate = 0.01



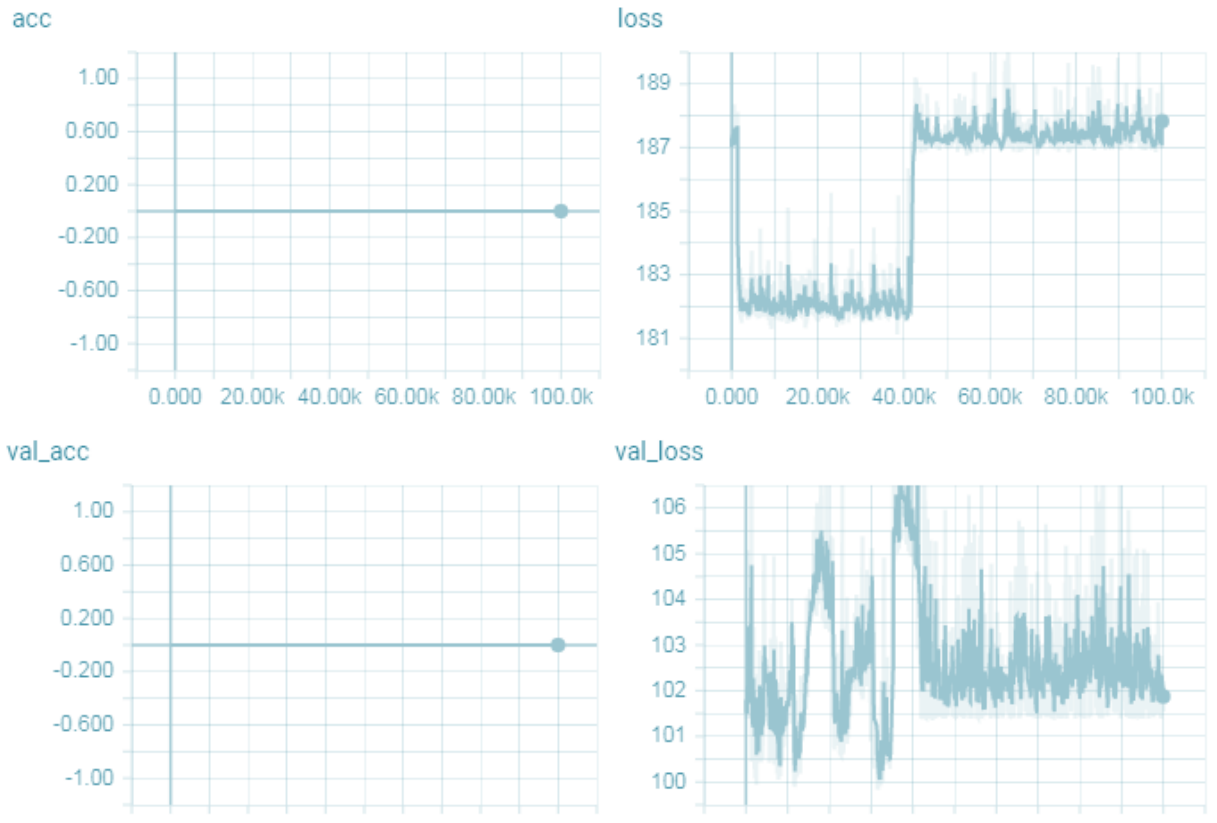
Dla struktury 30-10-1, learning rate = 0.1



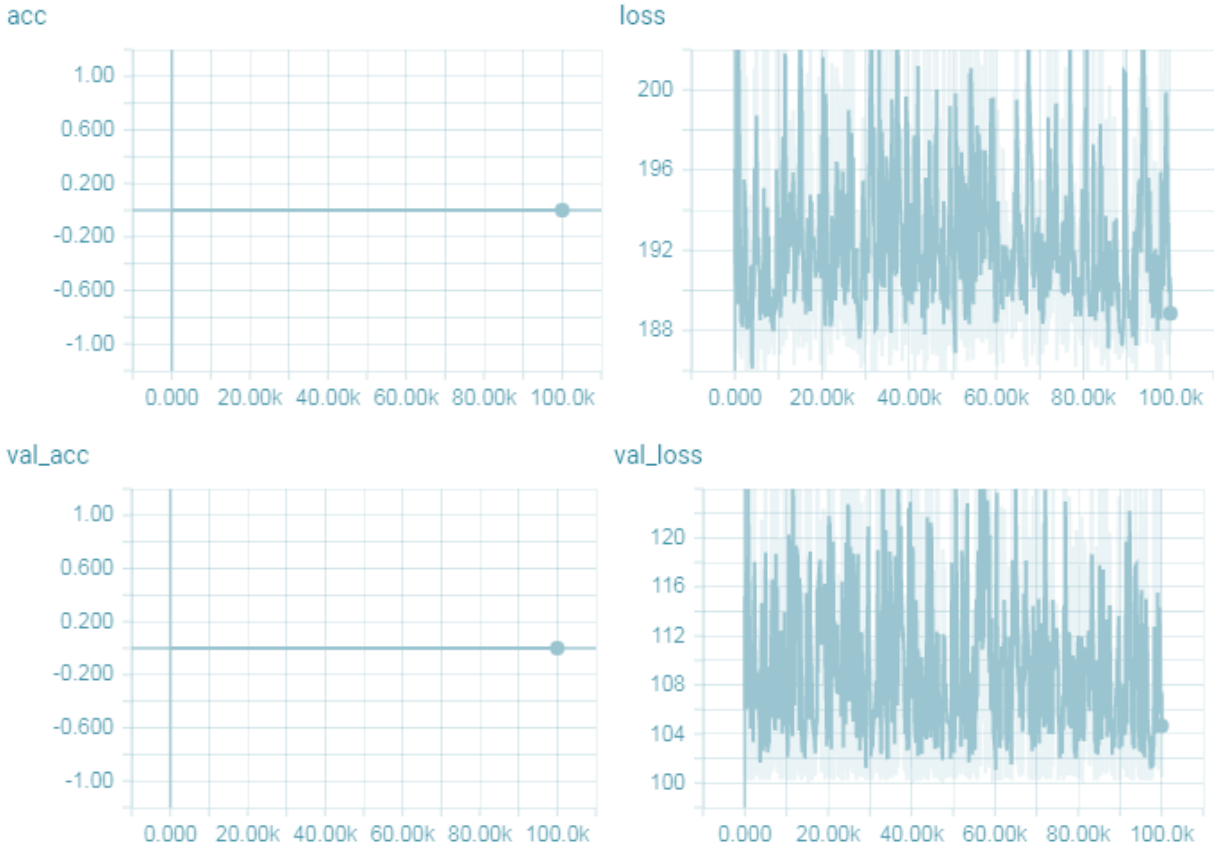
Dla struktury 30-10-1, learning rate = 0.4



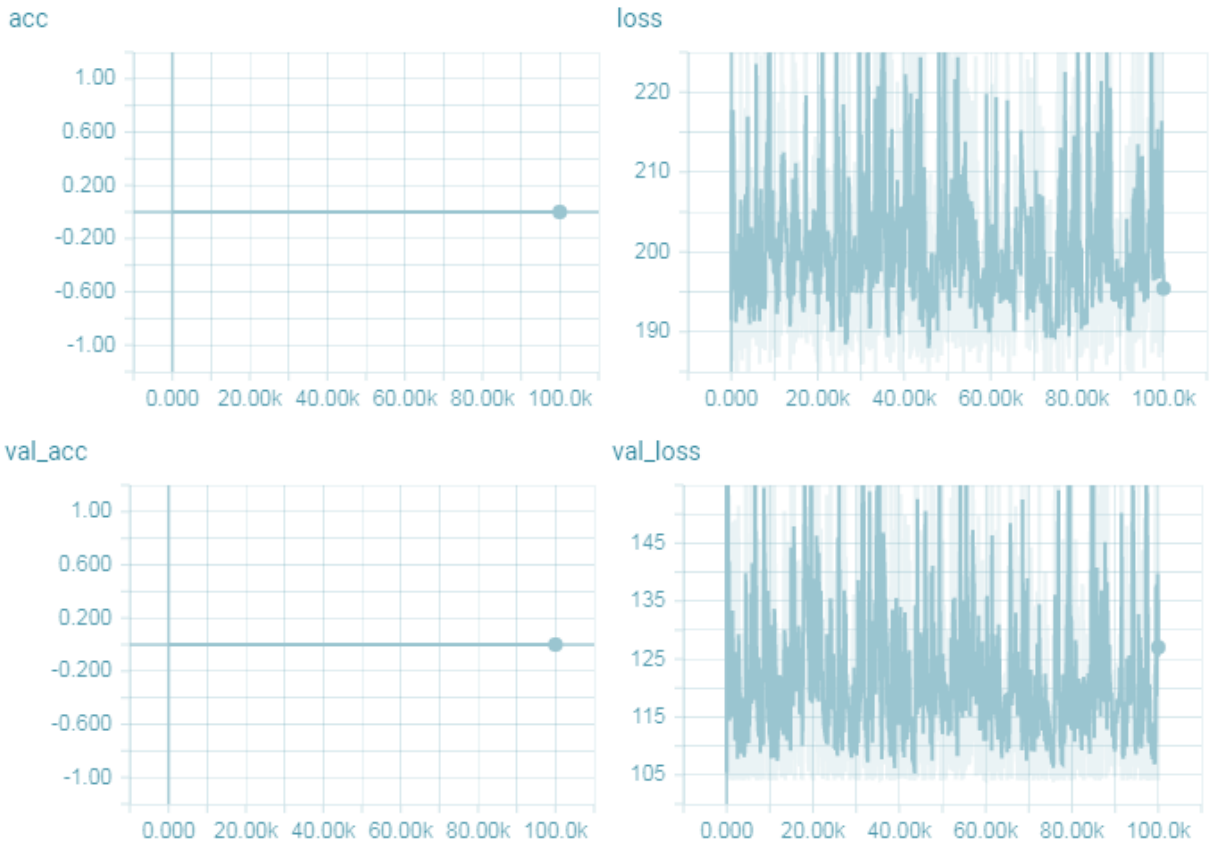
Dla struktury 30-30-30-1, learning rate = 0.01



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



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



4. 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.

5. 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.

6. Listing kodu

LITERKI.PY

```
from
pybrain3.datasets
import
SupervisedDataSet
                    daneWejsciowe = SupervisedDataSet(35, 20)
                    daneWejsciowe.addSample((
                        -1, 1, 1, 1, -1,
                        1, -1, -1, -1, 1,
                        1, -1, -1, -1, 1,
                        1, 1, 1, 1, 1,
                        1, -1, -1, -1, 1,
                        1, -1, -1, -1, 1,
                        1, -1, -1, -1, 1
                    ),
                        (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
```

```
daneWejsciowe.addSample((
    1, 1, 1, 1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, 1, 1, 1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, 1, 1, 1, -1
),
    (0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    -1, 1, 1, 1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, 1,
    -1, 1, 1, 1, -1
),
    (0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, 1, 1, 1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, 1, 1, 1, -1
),
    (0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, 1, 1, 1, 1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, 1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, 1
),
    (0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, 1, 1, 1, 1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, -1, -1,
    1, -1, -1, -1, -1,
```

```
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1
),
    (0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    -1, 1, 1, 1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, -1,
    1, -1, 1, 1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    -1, 1, 1, 1, -1
),
    (0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, 1, 1, 1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1
),
    (0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, 1, 1, 1, -1,
    -1, -1, -1, 1, -1,
    -1, -1, -1, 1, -1,
    -1, -1, -1, 1, -1,
    -1, -1, -1, 1, -1,
    1, -1, -1, 1, -1,
    -1, 1, 1, -1, -1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, -1, -1, -1, 1,
```

```
    1, -1, -1, 1, -1,
    1, -1, 1, -1, -1,
    1, 1, -1, -1, -1,
    1, -1, 1, -1, -1,
    1, -1, -1, 1, -1,
    1, -1, -1, -1, 1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    -1, 1, 1, 1, -1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, -1, -1, -1, 1,
    1, 1, -1, 1, 1,
    1, -1, 1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, -1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, 1, 1, 1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, 1, 1, 1, 1
```

```
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, 1, 1, -1, -1,
    1, -1, -1, 1, -1,
    1, -1, -1, 1, -1,
    1, 1, 1, -1, -1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0))
daneWejsciowe.addSample((
    1, 1, 1, -1, -1,
    1, -1, -1, 1, -1,
    1, -1, -1, 1, -1,
    1, 1, 1, -1, -1,
    1, -1, 1, -1, -1,
    1, -1, -1, 1, -1,
    1, -1, -1, -1, 1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0))
daneWejsciowe.addSample((
    1, 1, 1, 1, 1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0))
daneWejsciowe.addSample((
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, 1, -1, 1,
    1, 1, -1, 1, 1,
    1, -1, -1, -1, 1
),
    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0))
daneWejsciowe.addSample((
    1, 1, 1, 1, 1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
```

```
                    1, 1, 1, 1, 1,
                    -1, -1, -1, -1, 1,
                    -1, -1, -1, -1, 1,
                    1, 1, 1, 1, 1
                ),
                    (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1))
```

PropagTrener.PY

```
from
pybrain3.supervised.trainers
import BackpropTrainer
                            from literki import daneWejsciowe
                            import literki
                            import siec
                            litery = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J",
                            "K", "U", "M", "L", "O", "P", "R", "T", "W", "S"]
                            inp = daneWejsciowe['input']
                            trener = BackpropTrainer(siec.siec,
                            dataset=literki.daneWejsciowe, learningrate=0.1)
                            trener.trainEpochs(1000)
                            for i in range(20):
                                print(litery[i])
                                temp = siec.siec.activate(inp[i])
                                for j in range(20):
                                    print(temp[j])
                            print("\n")
```

SIEC.PY

```
from
pybrain3
import *
            siec = FeedForwardNetwork()
            wejscie = LinearLayer(35)
            ukryty = SigmoidLayer(30)
            wyjscie = LinearLayer(20)
            bias = BiasUnit()
            siec.addInputModule(wejscie)
            siec.addModule(bias)
            siec.addModule(ukryty)
            siec.addOutputModule(wyjscie)
```

```
bias_ukryty = FullConnection(bias, ukryty)
wejscie_ukryty = FullConnection(wejscie, ukryty)
ukryty_wyjscie = FullConnection(ukryty, wyjscie)
siec.addConnection(bias_ukryty)
siec.addConnection(wejscie_ukryty)
siec.addConnection(ukryty_wyjscie)
siec.sortModules()
```