

Sprawozdanie
Z przedmiotu: Uczenie maszynowe
Ćwiczenie 1

Wykonały:

Anna Vezdenetska nr albumu: 107938

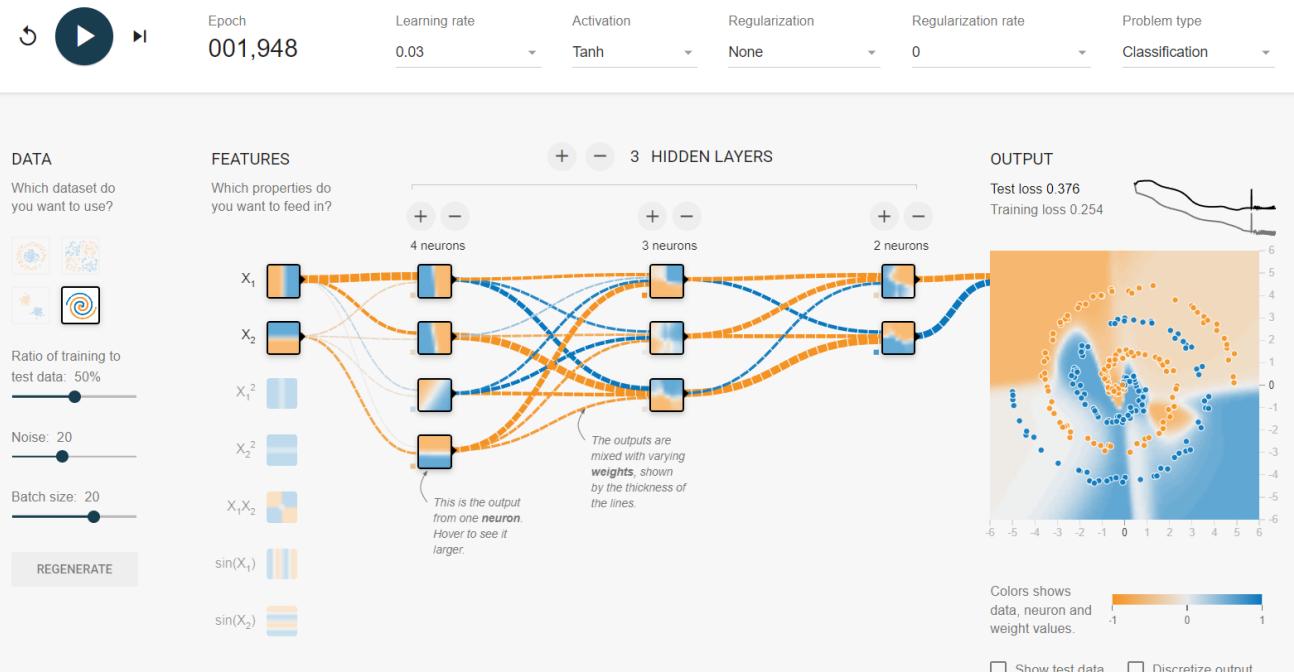
Aleksandra Kot nr albumu: 104061

Rozdział 1 – Playground

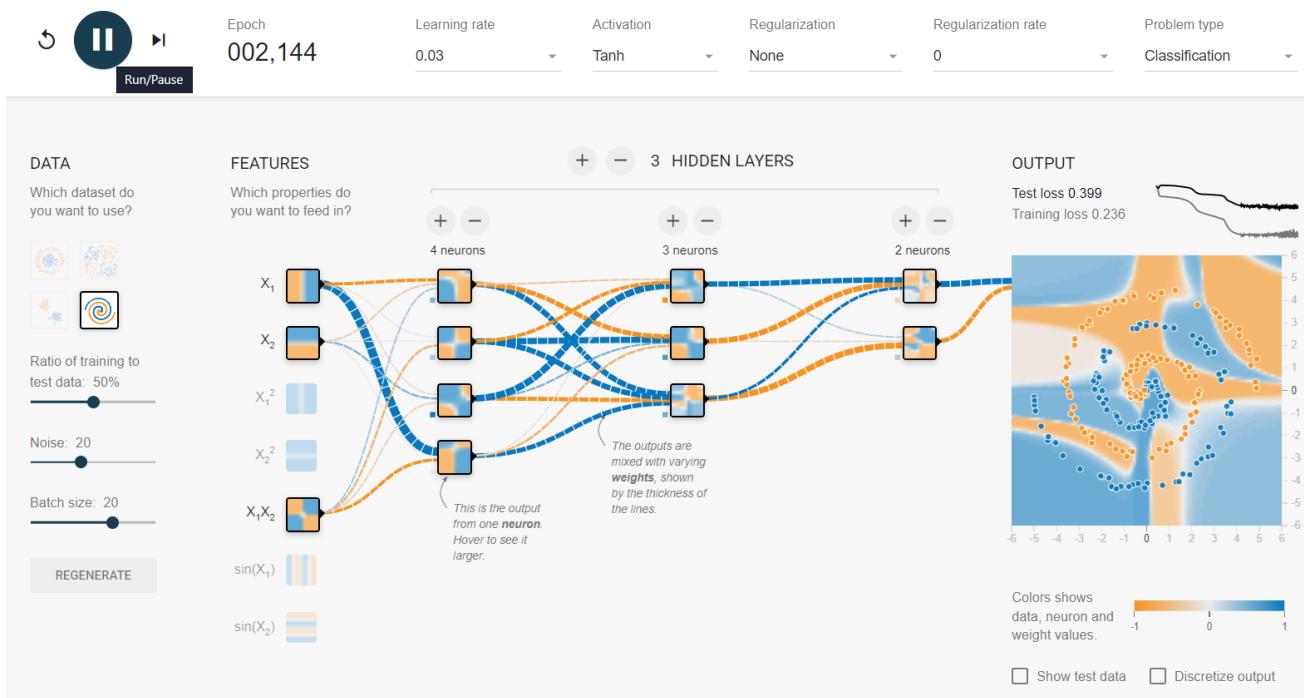
Na podstawie <https://playground.tensorflow.org> wykonać następujące zadania:

- Przeprowadzić badania 7 różnych architektur sieci (różne ilości ukrytych warstw, zmiana ilości neuronów).

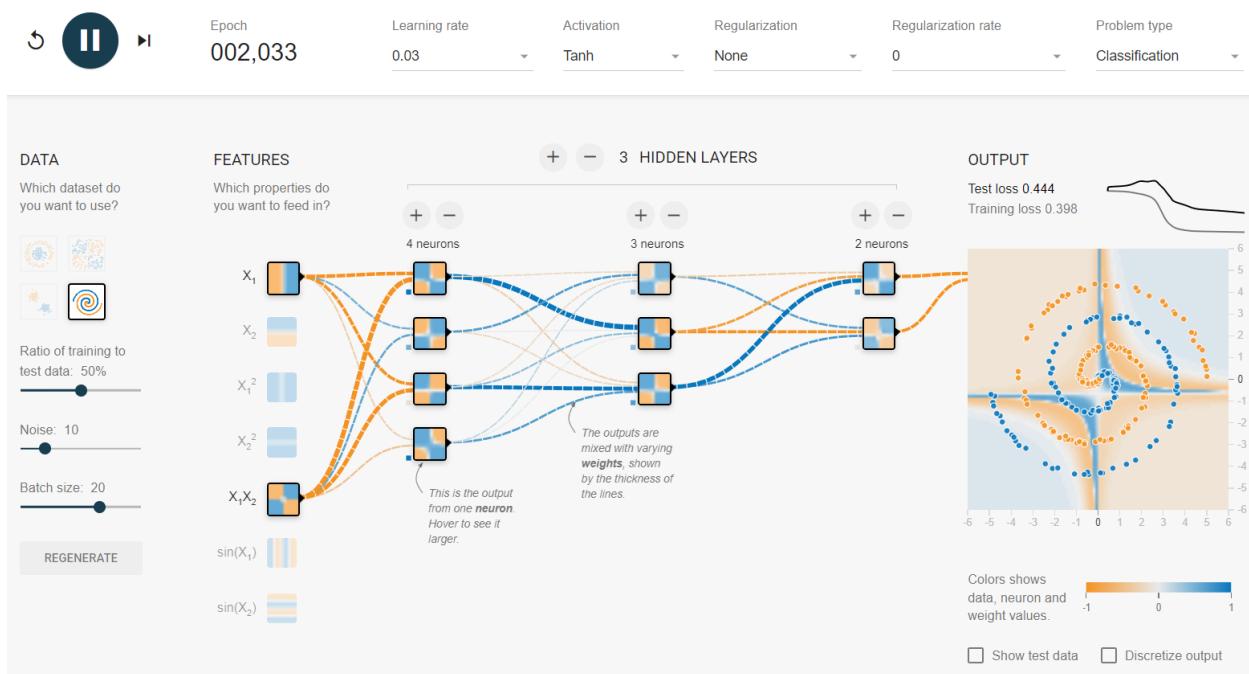
I. Architektura sieci



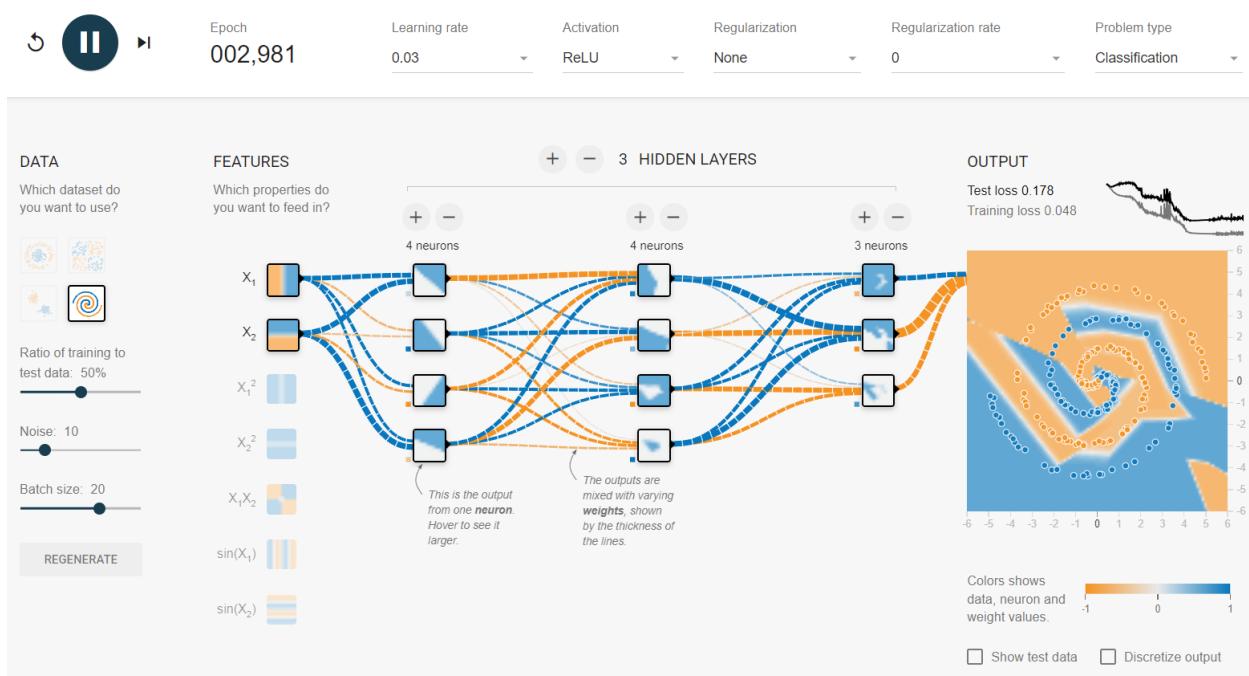
II. Architektura sieci



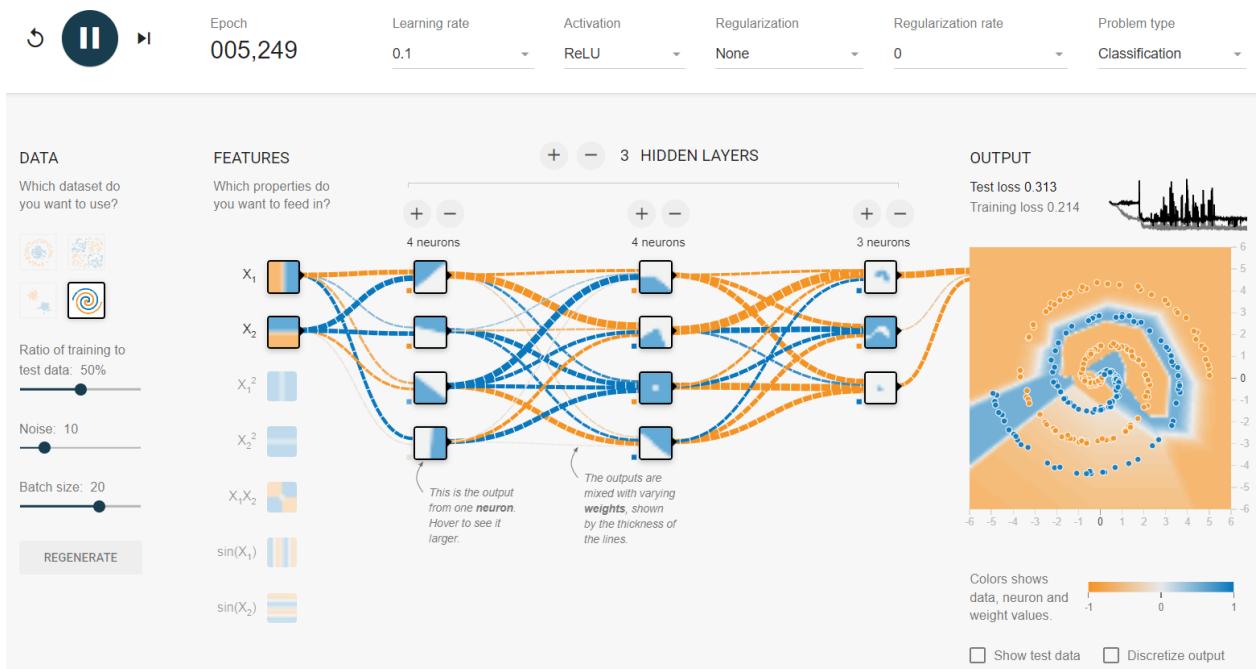
III. Architektura sieci



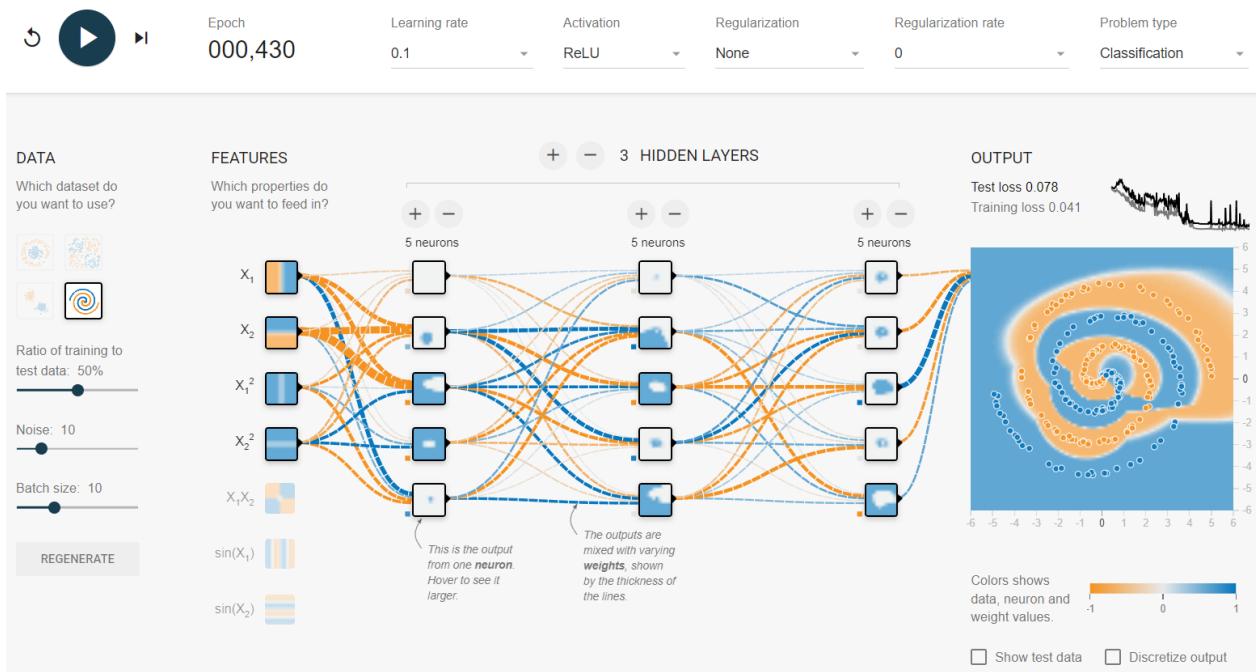
IV. Architektura sieci



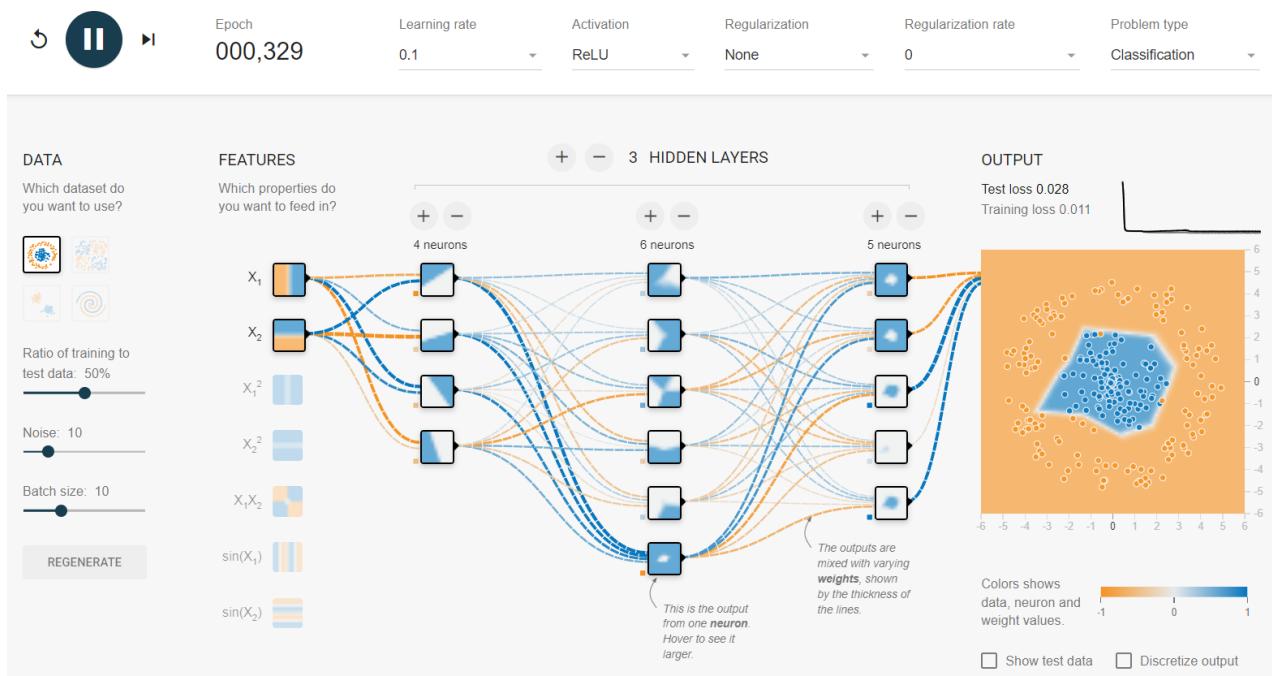
V. Architektura sieci



VI. Architektura sieci



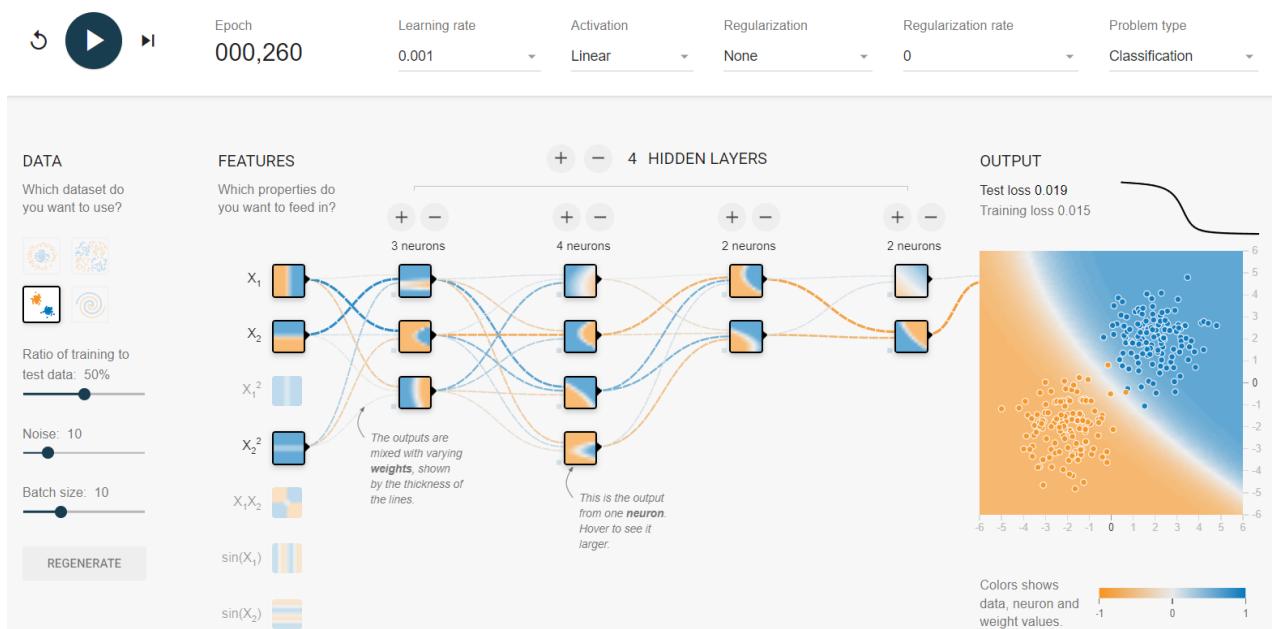
VII. Architektura sieci

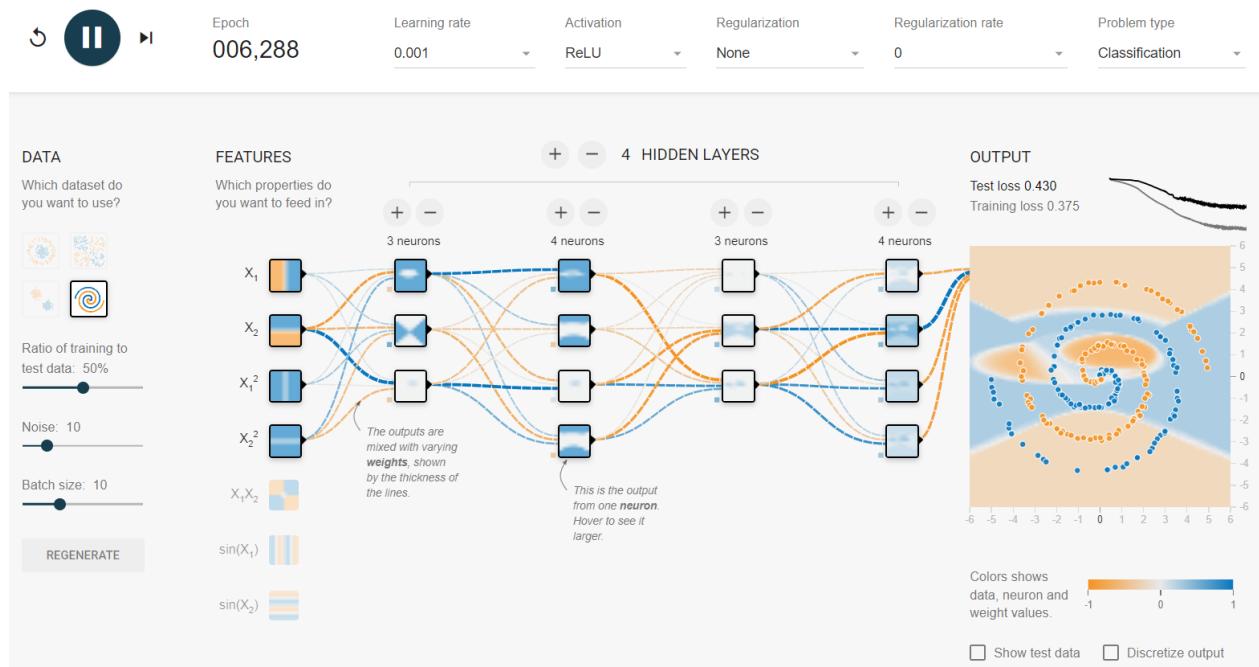


2. Sprawdzić wpływ learning rate, activation, regularization regularization rate dla problemu klasyfikacji i dwóch różnych typów zbiorów.

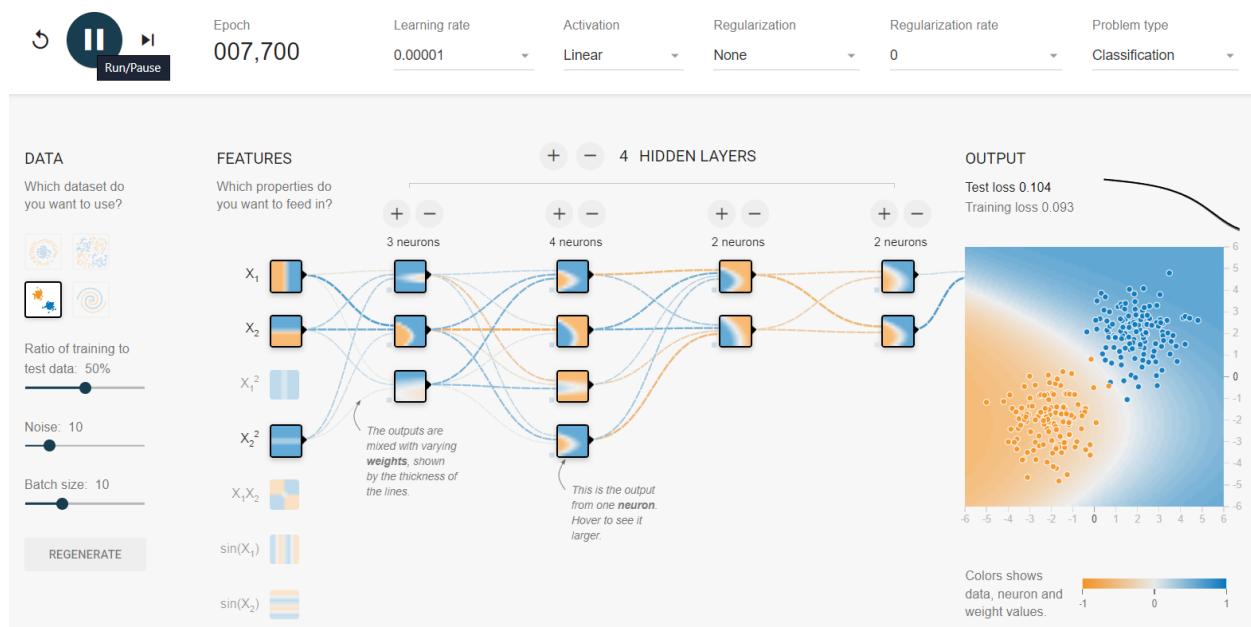
2.1. Wpływ learning rate Gausian/Spiral

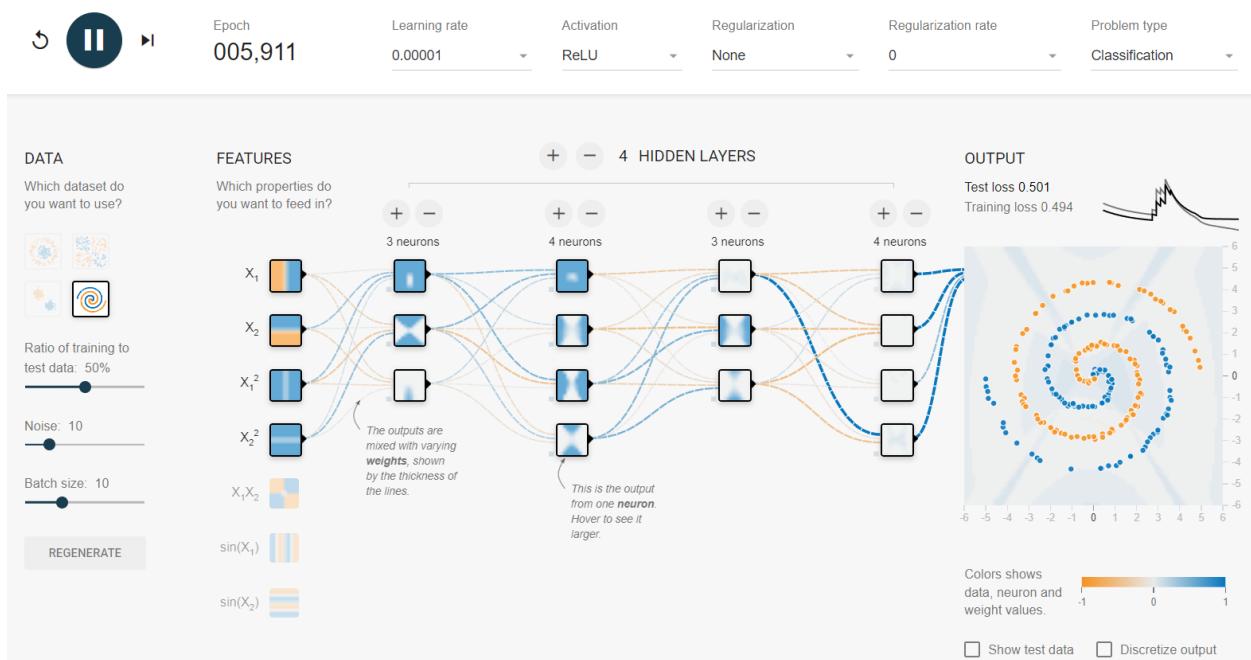
2.1.1. Learning rate 0.001 Gausian/Spiral



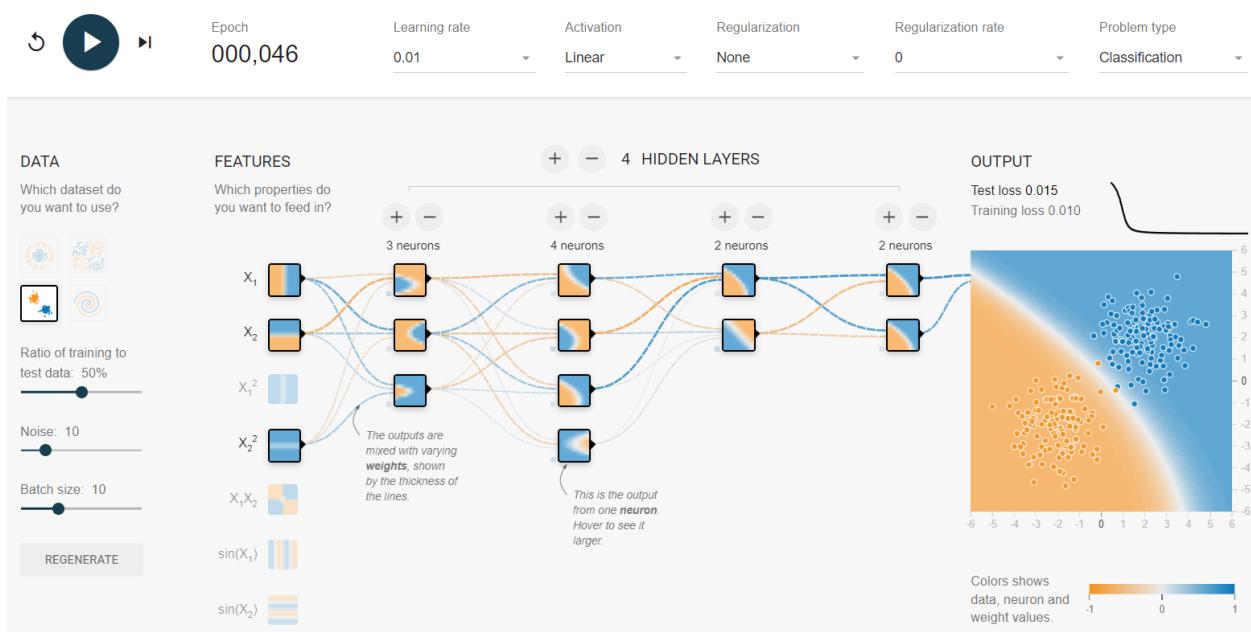


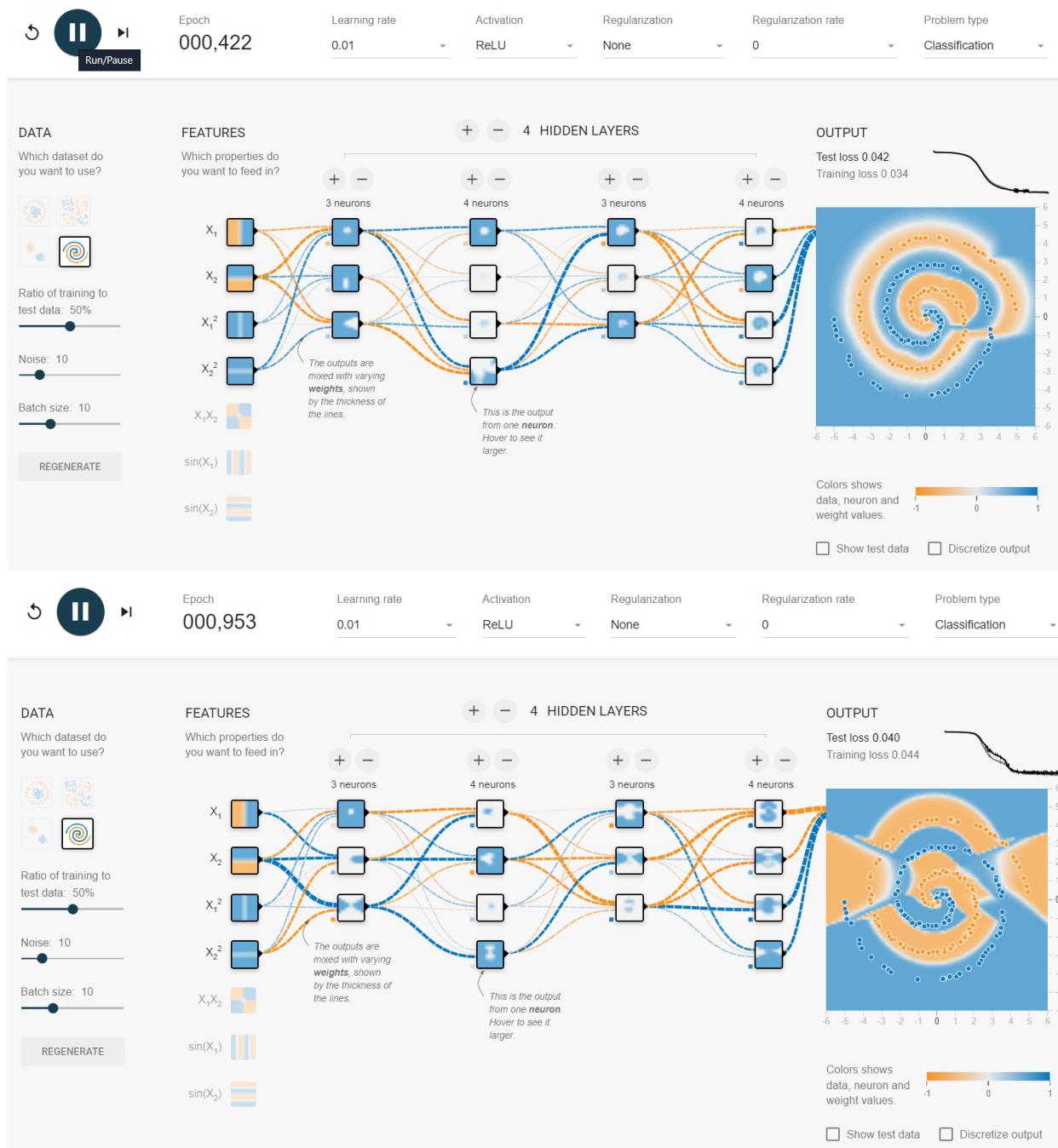
2.1.2. Learning rate 0.00001Gaussian/Spiral



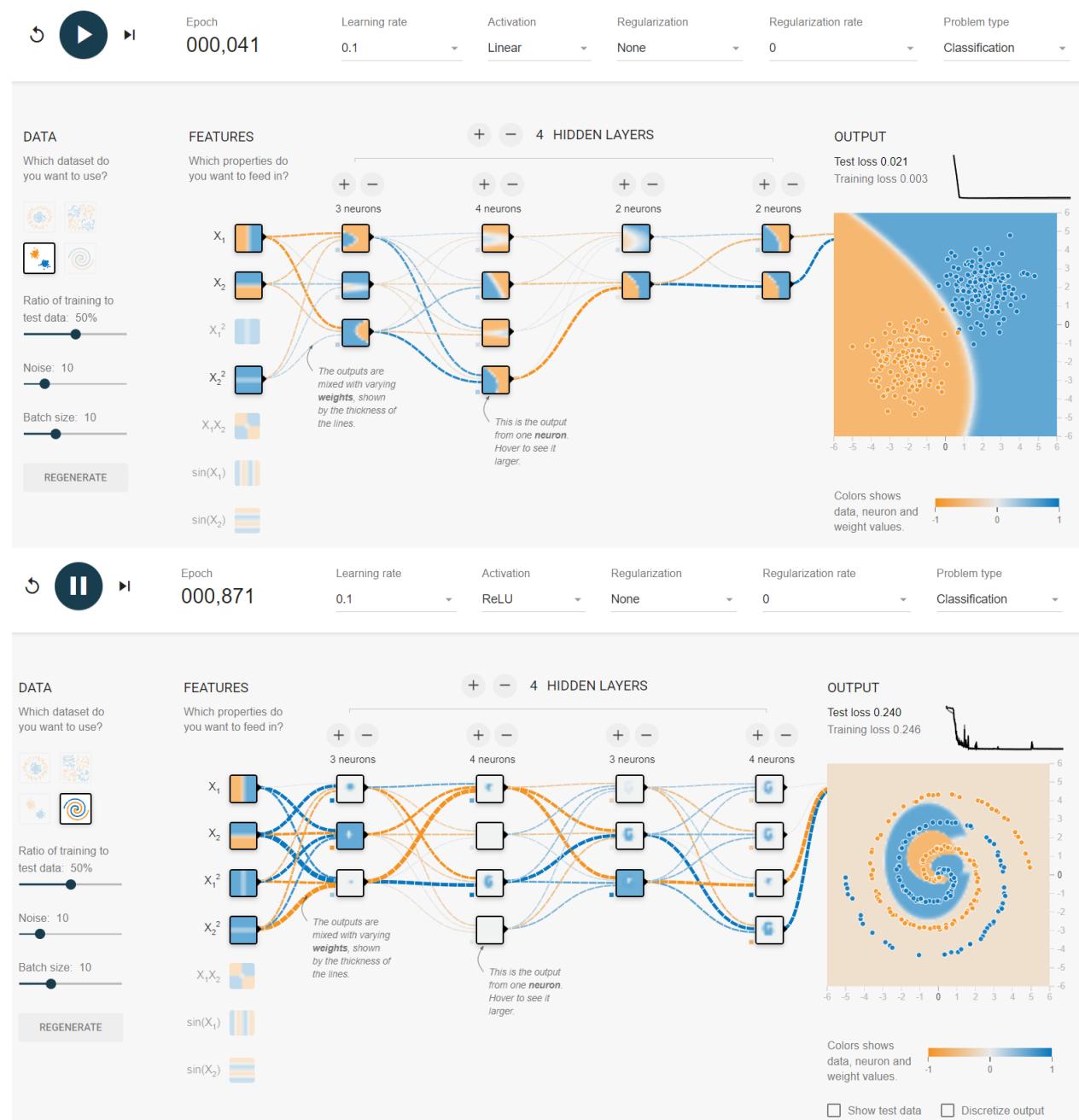


2.1.3. Learning rate 0.01 Gausian/Spiral





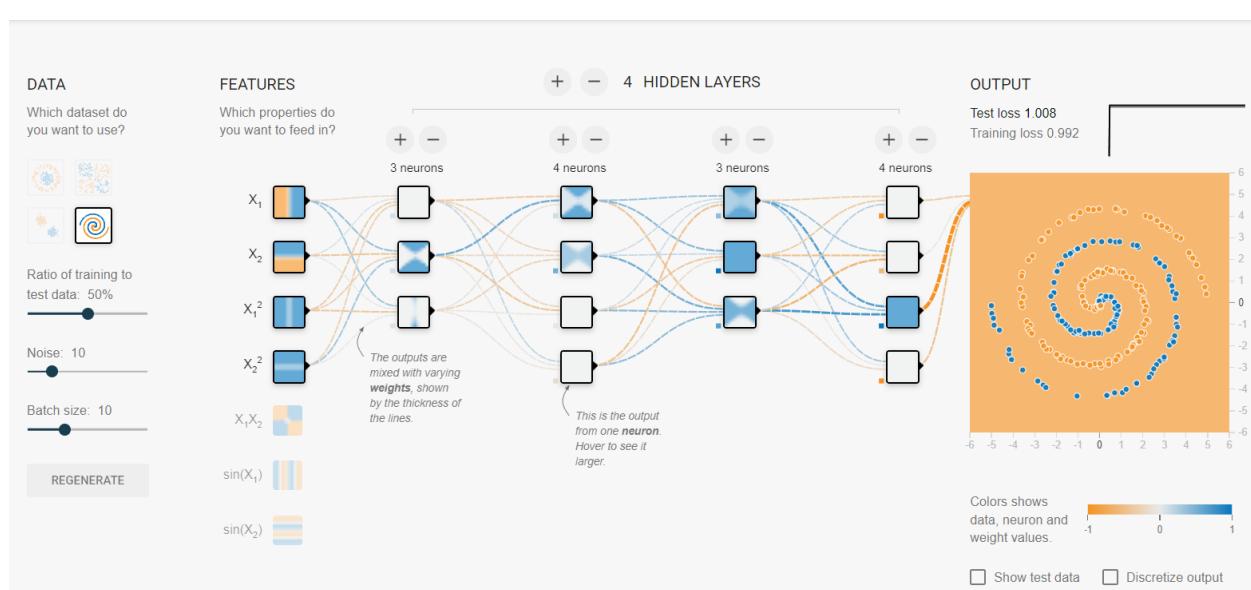
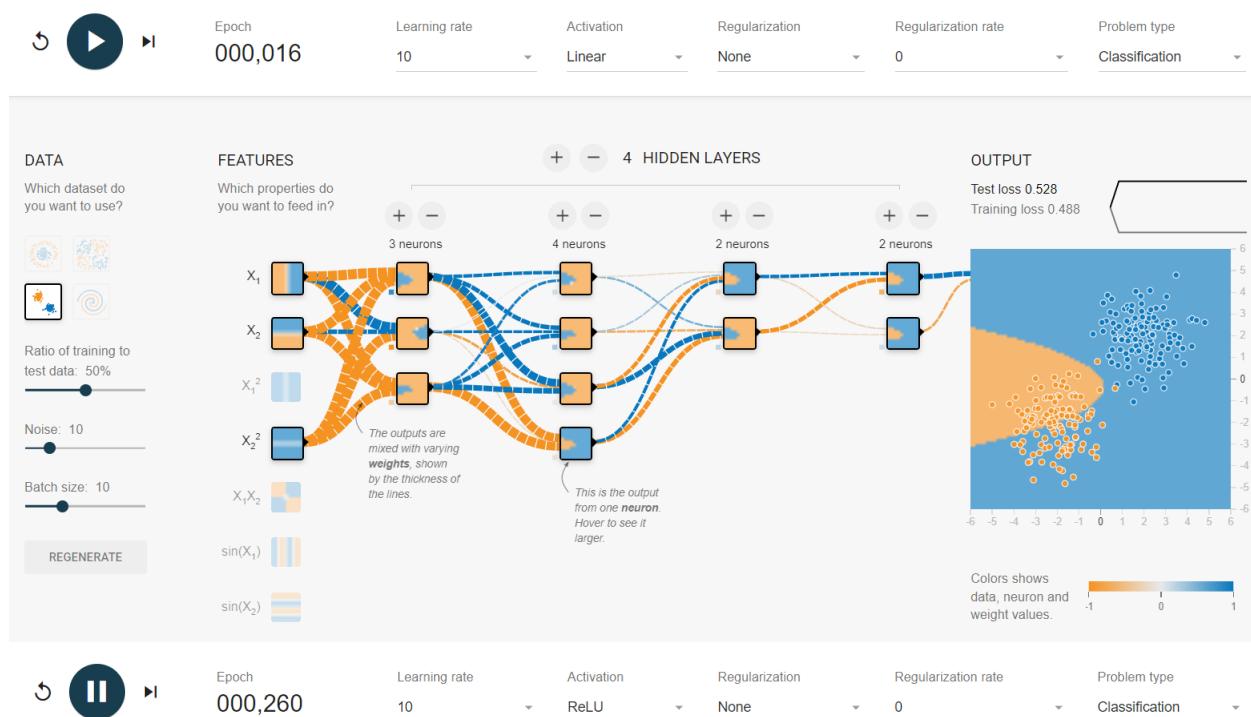
2.1.4. Learning rate 0.1



2.1.5. Learning rate 1

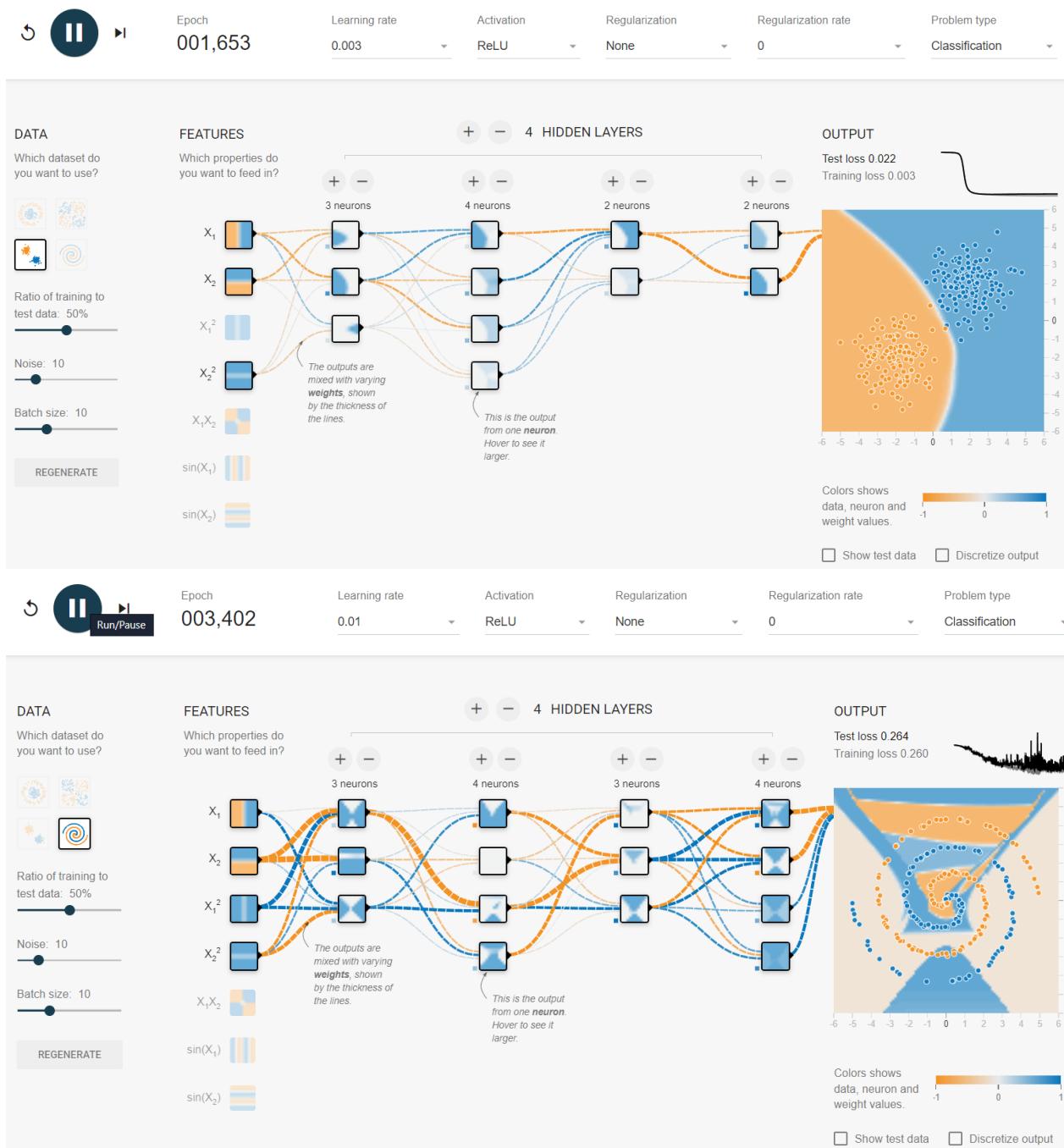


2.1.6. Learning rate 10

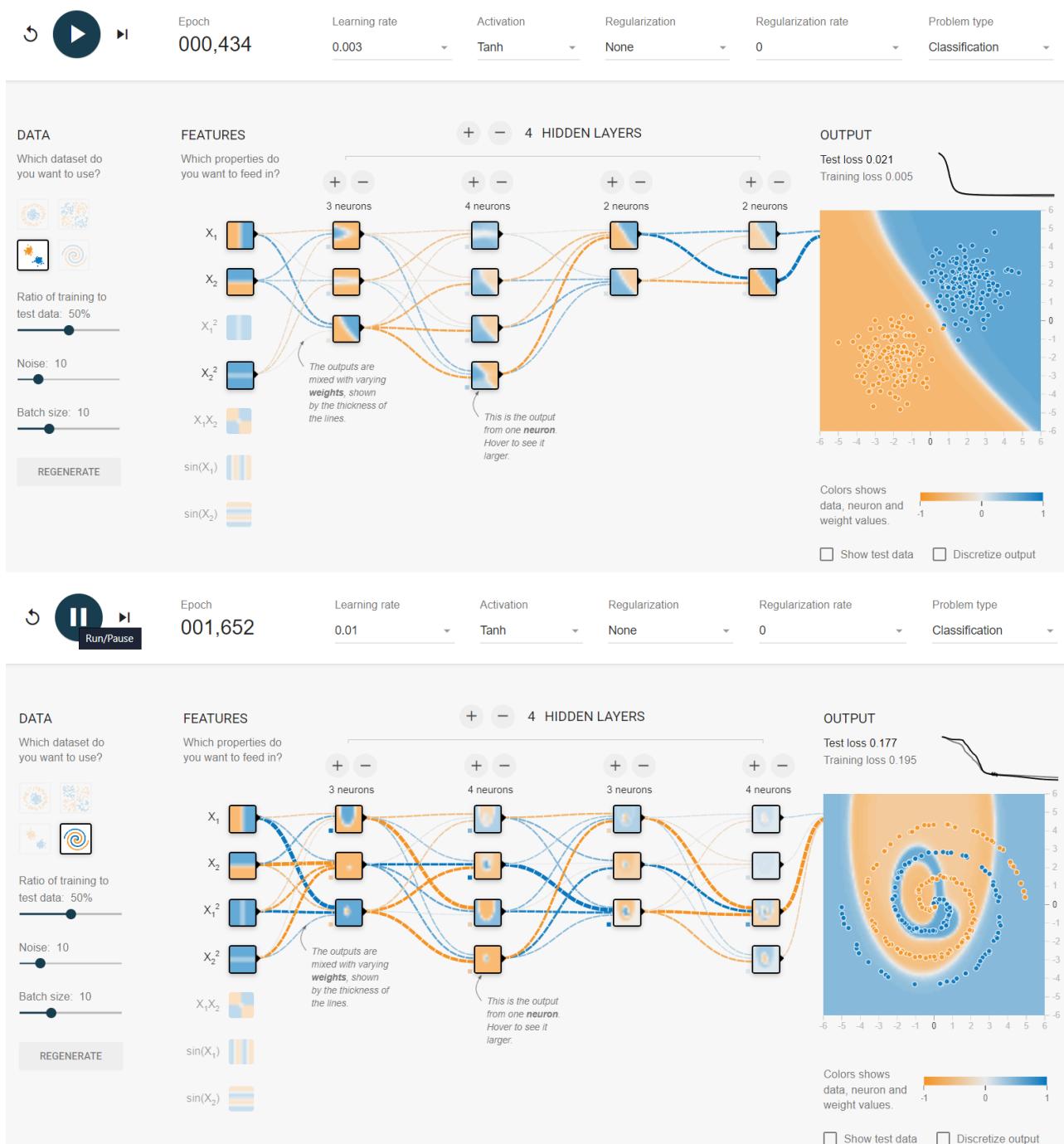


2.2. Activation Gausian/Spiral

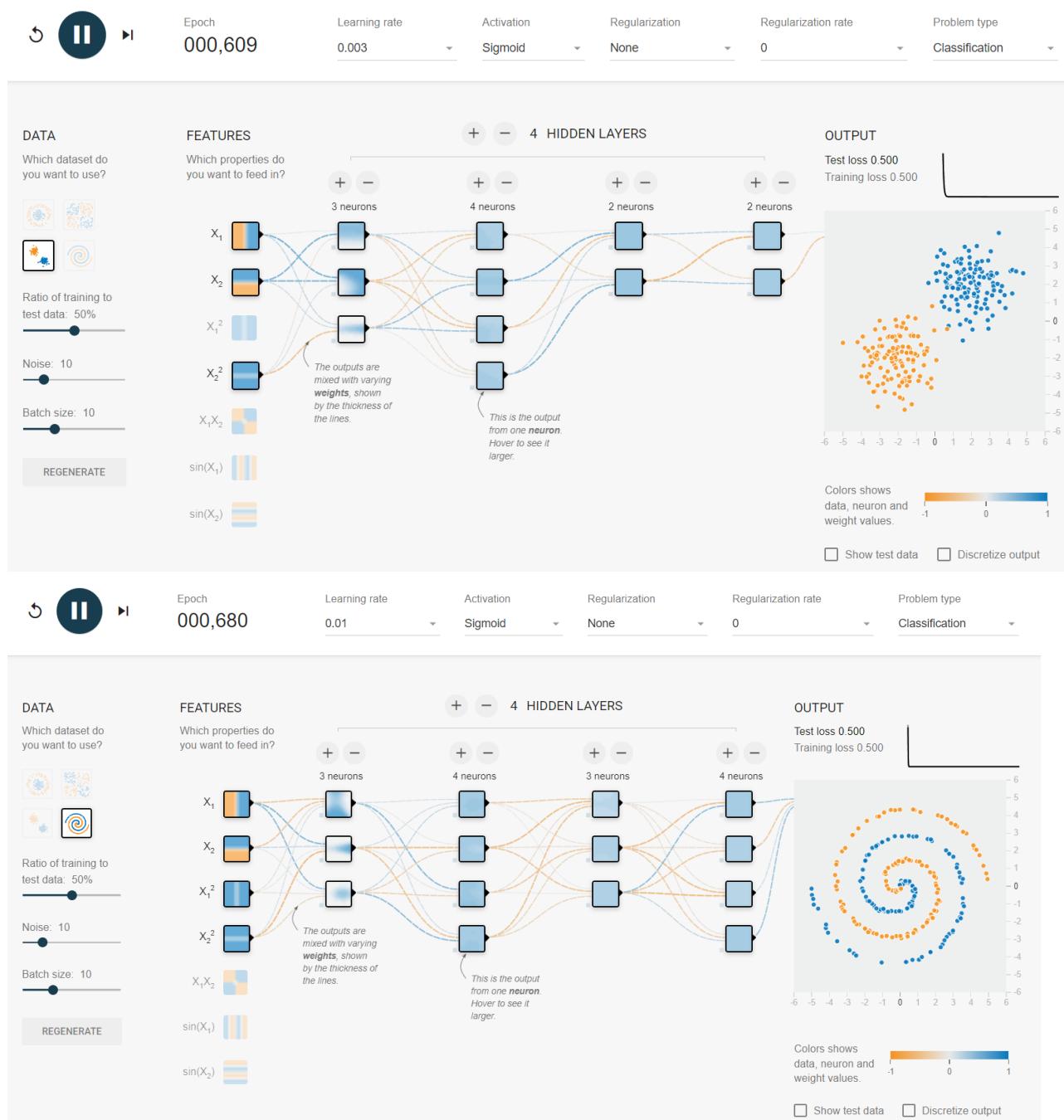
2.2.1. Activation ReLU



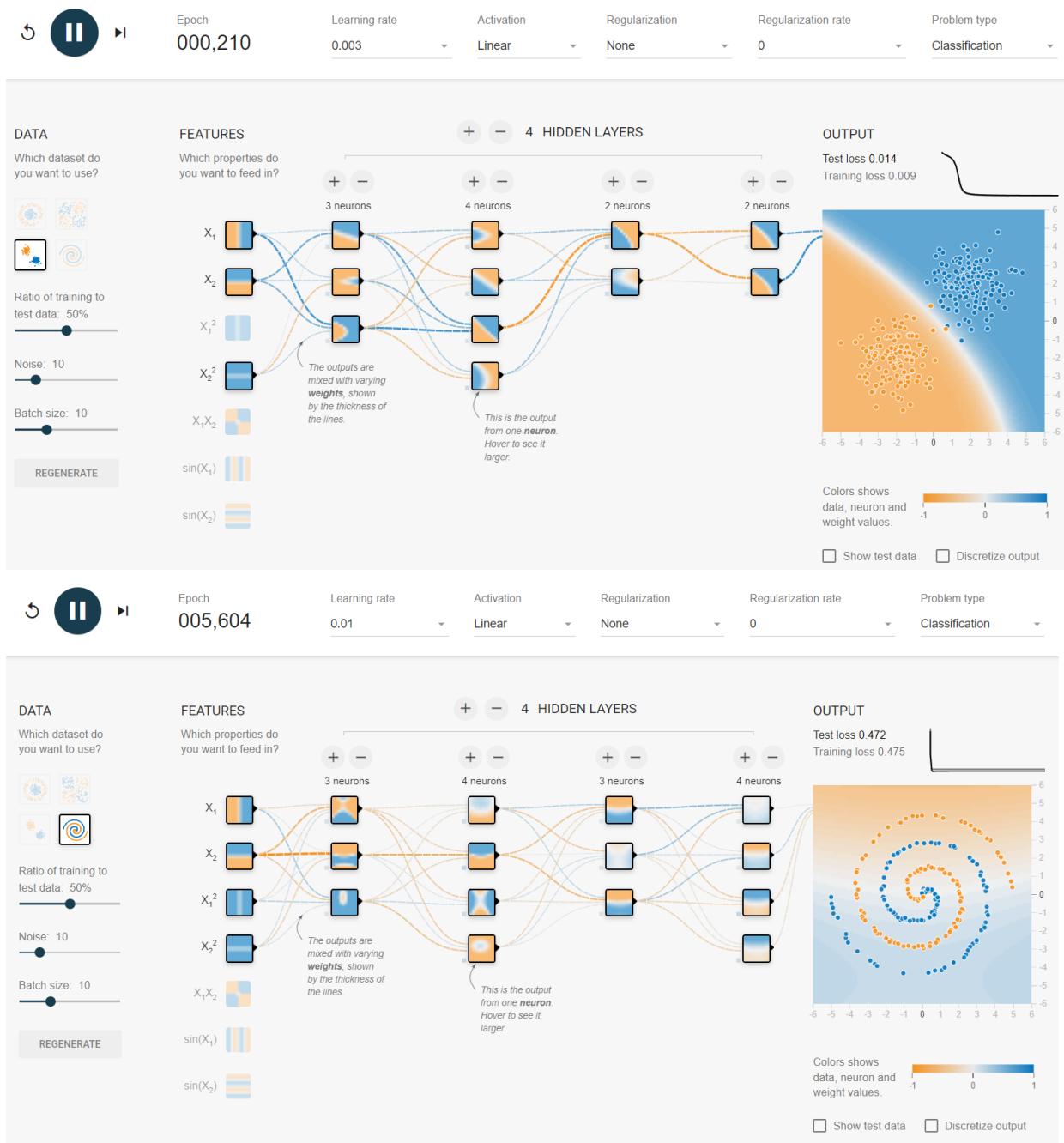
2.2.2. Activation Tanh



2.2.3. Activation Sigmoid



2.2.4. Activation Linear

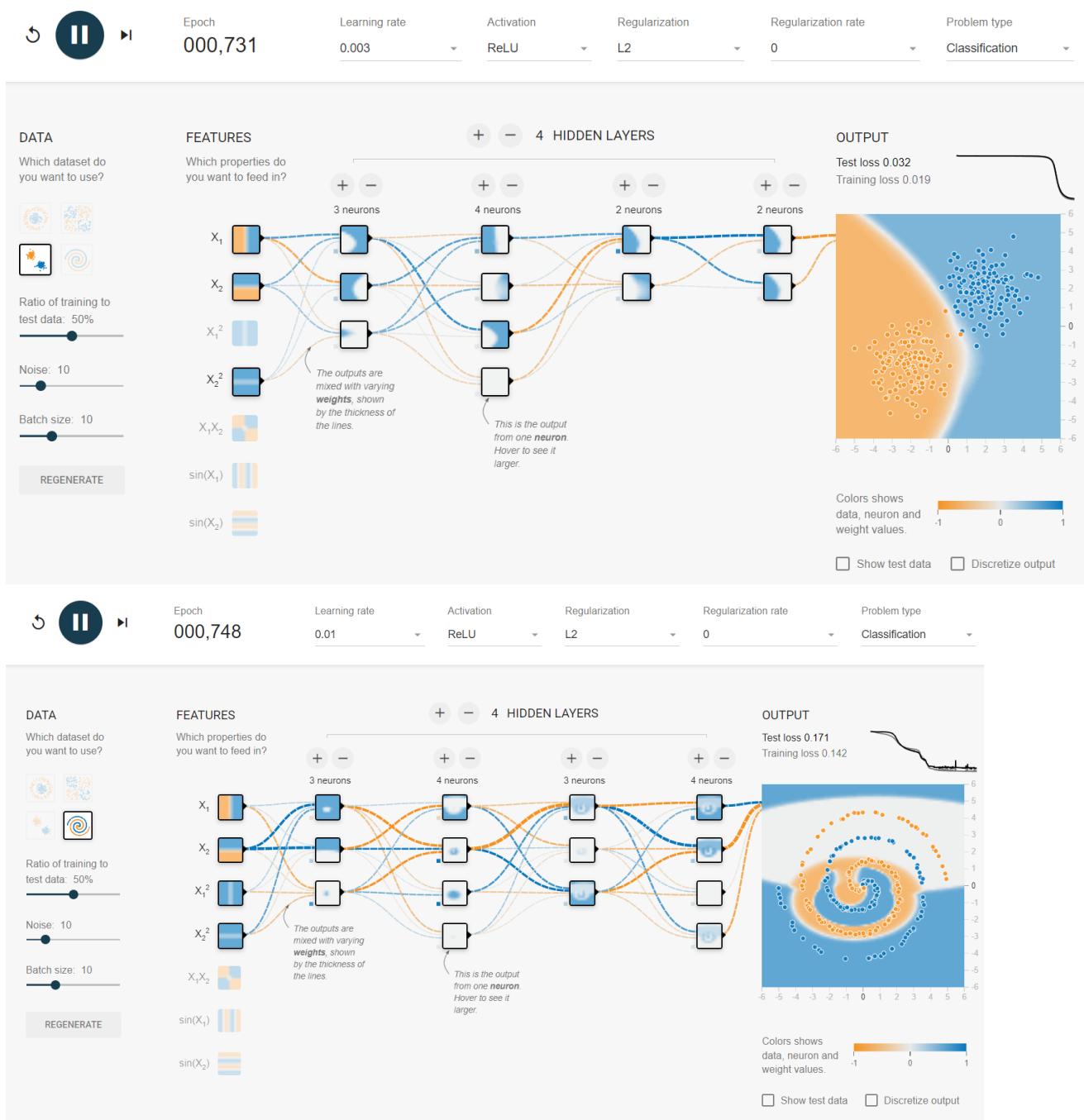


2.3. Regulation Gausian/Spiral

2.3.1. Regularization L1

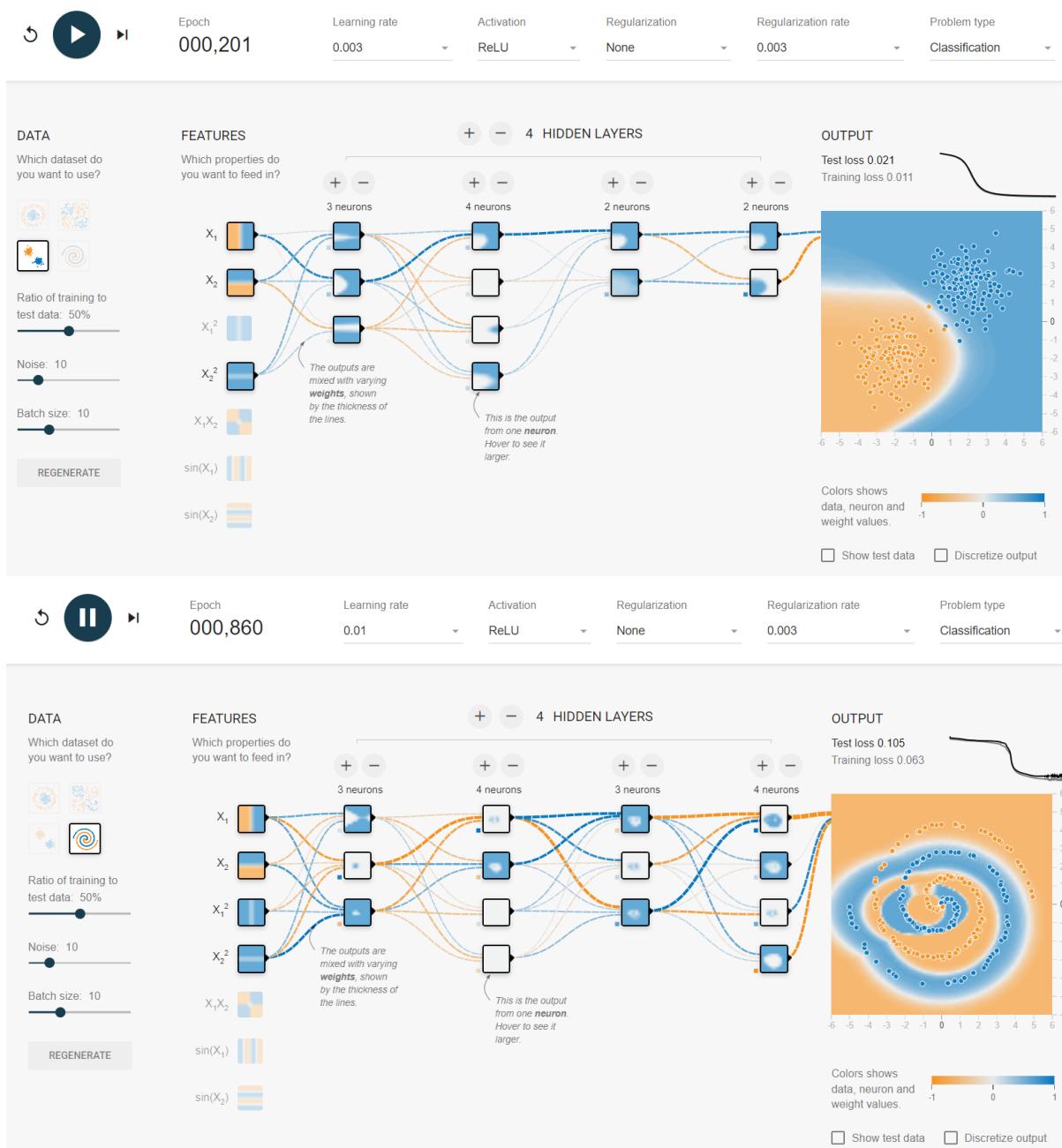


2.3.2. Regulation L2



2.4. Regulation rate Gaussian/Spiral

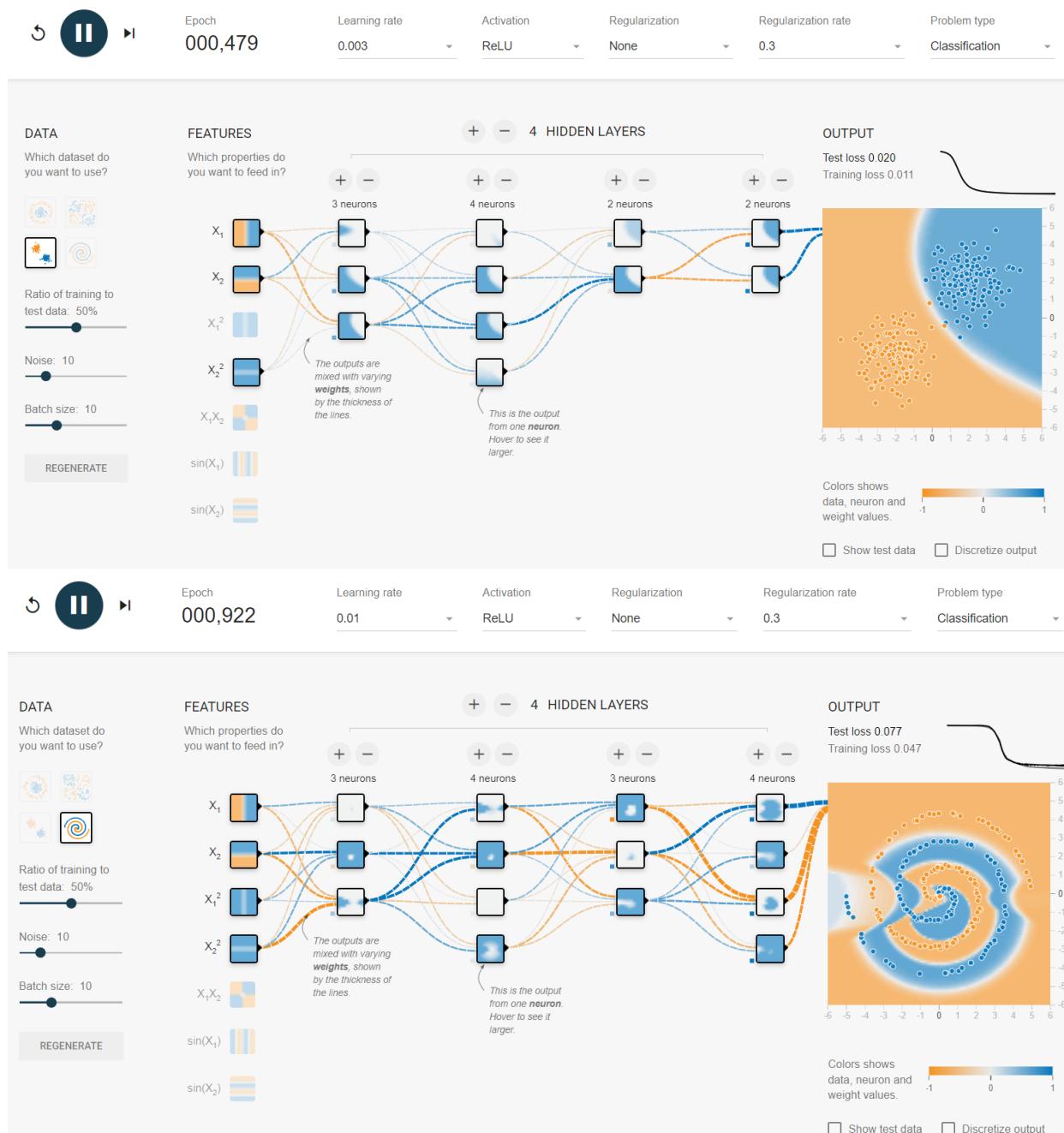
2.4.1. Regularization rate 0.003



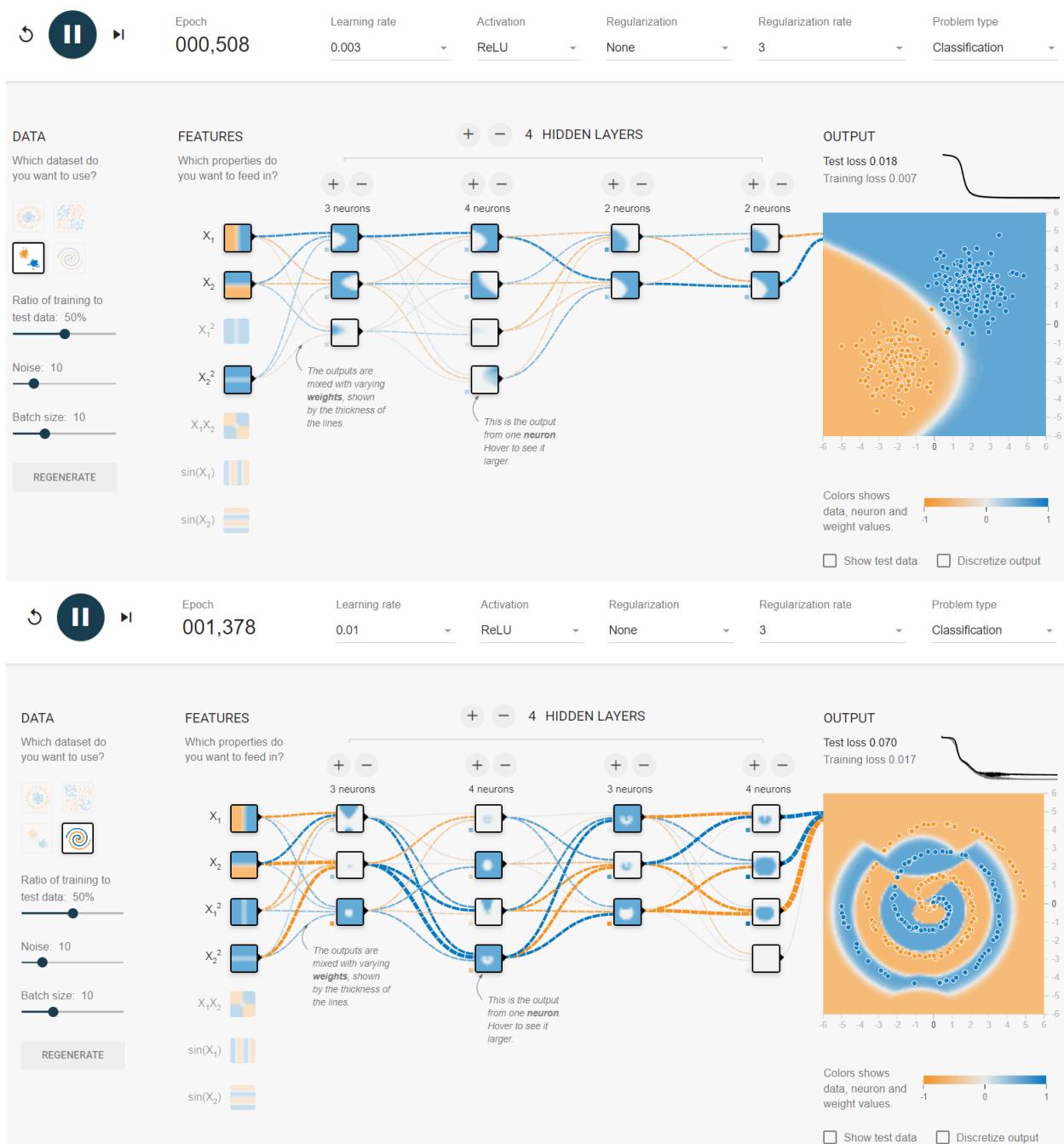
2.4.2. Regularization rate 0.03



2.4.3. Regularization rate 0.3



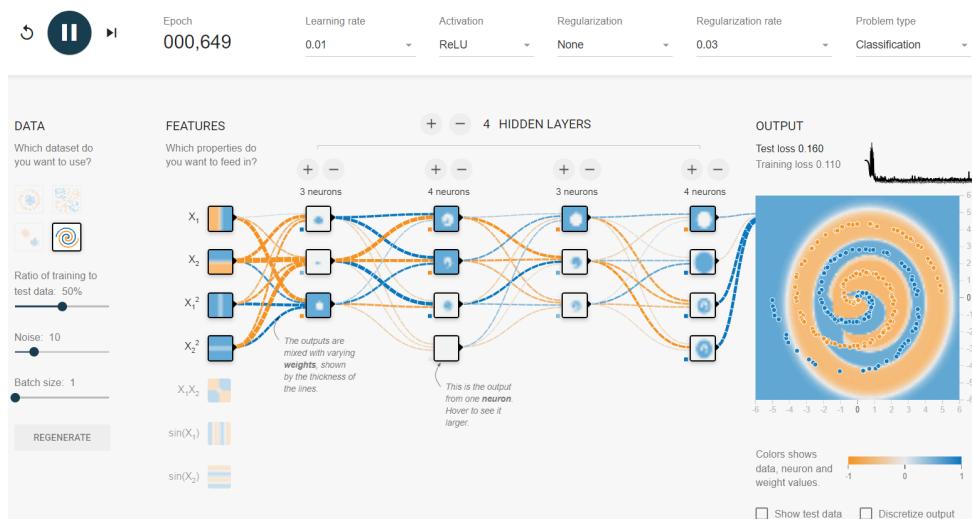
2.4.4. Regularization rate 3

**Wnioski:**

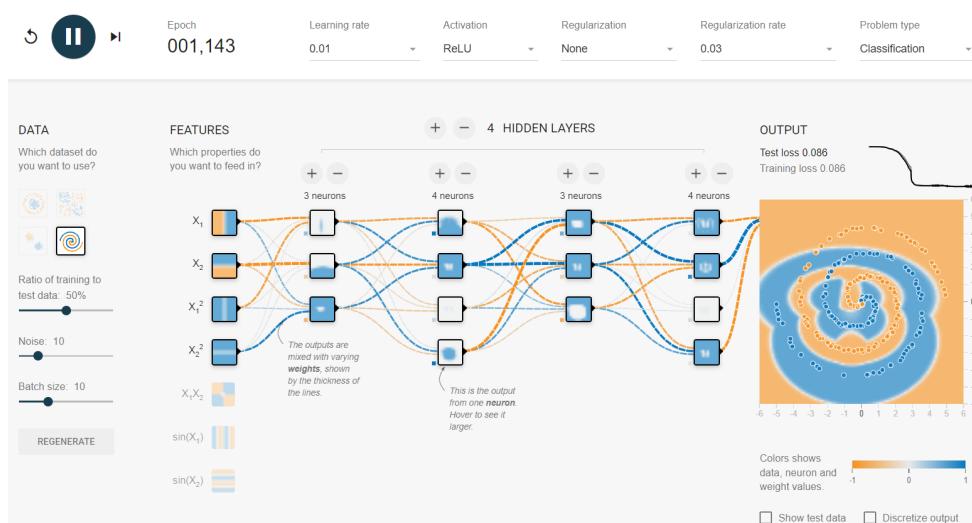
Podczas weryfikacji wpływu wartości zmiennych learning rate, activation, regularization oraz regularization rate, zauważono pewne prawidłowości, przykładowo dla otrzymania satysfakcyjujących wyników, ważne jest dobranie optymalnej wartości dla tych zmiennych. Różnica pomiędzy dwoma wybranymi typami zbiorów polega na tym że naruszyć pracę zbioru Gaussa jest niełatwo i tylko skrajnie wartości mogą doprowadzić do naruszenia pracy zbioru. Natomiast dostosowanie parametrów dla zbioru Spiral, jest dość skomplikowany, gdyż uzyskanie pożądanych wyników często nie jest możliwe do otrzymania.

3. Sprawdzić wpływ rozmiaru batch size.

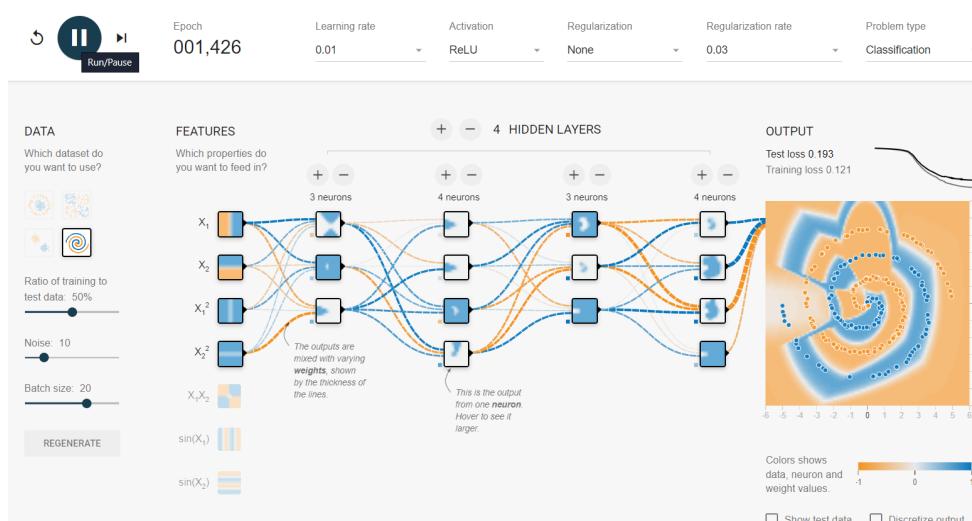
Batch size 1



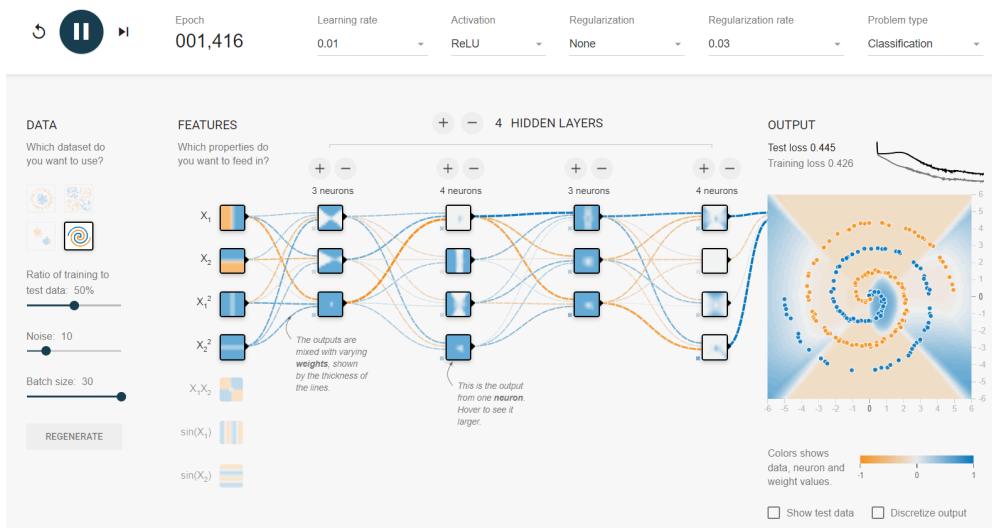
Batch size 10



Batch size 20



Batch size 30

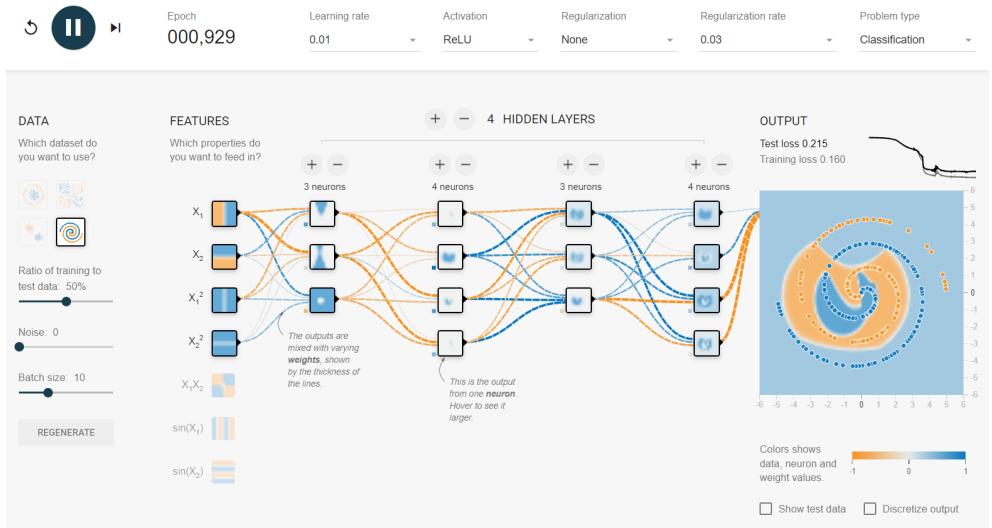


Wnioski:

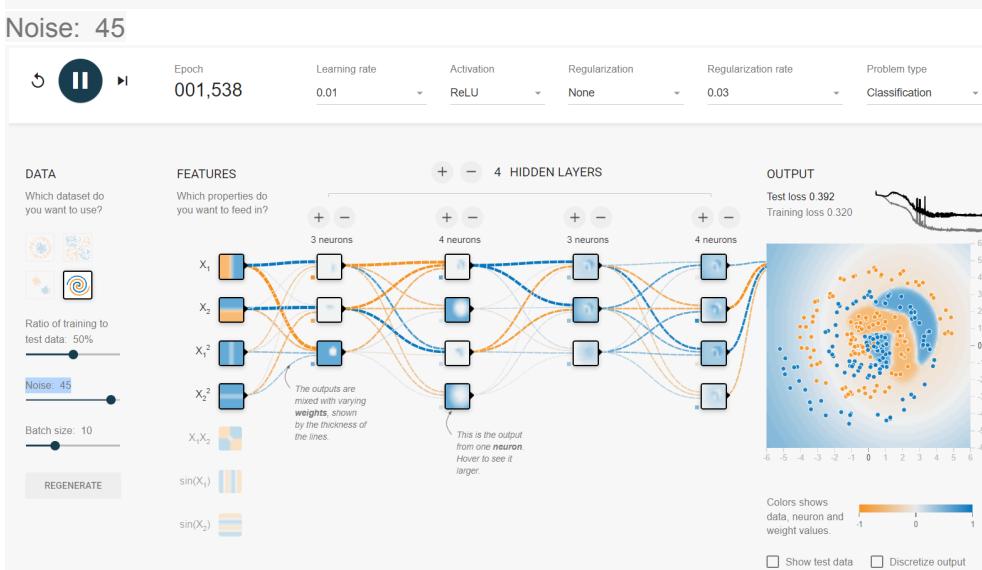
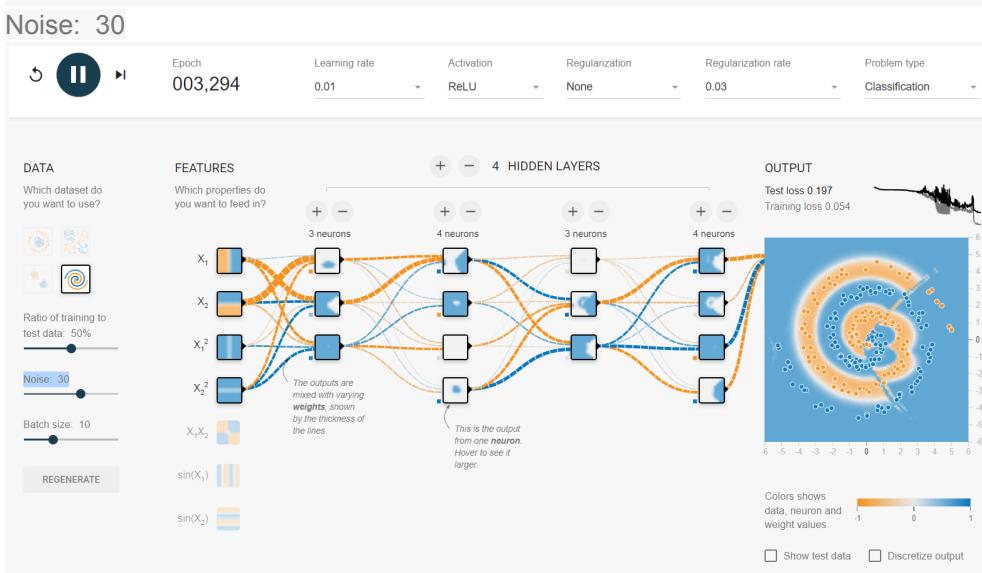
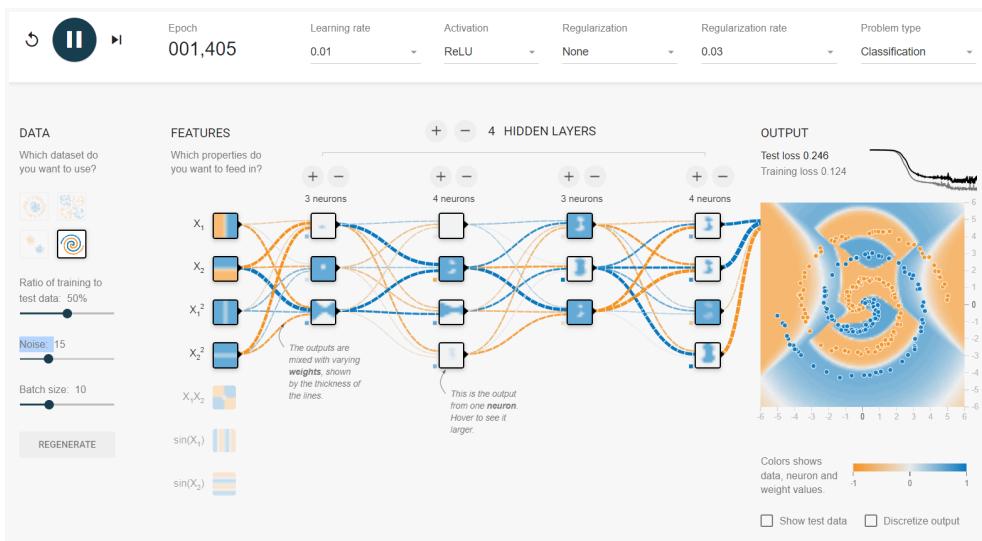
Dostosowywanie optymalnych wartości jest ważne dla każdego sprawdzanego parametru, który wpływa na prawidłowość funkcjonowania zbioru. Przy tym wielkość danego parametru sprawia że otrzymane wyniki znieksztalcają się.

4. Sprawdzić wpływ szumu.

Noise 0

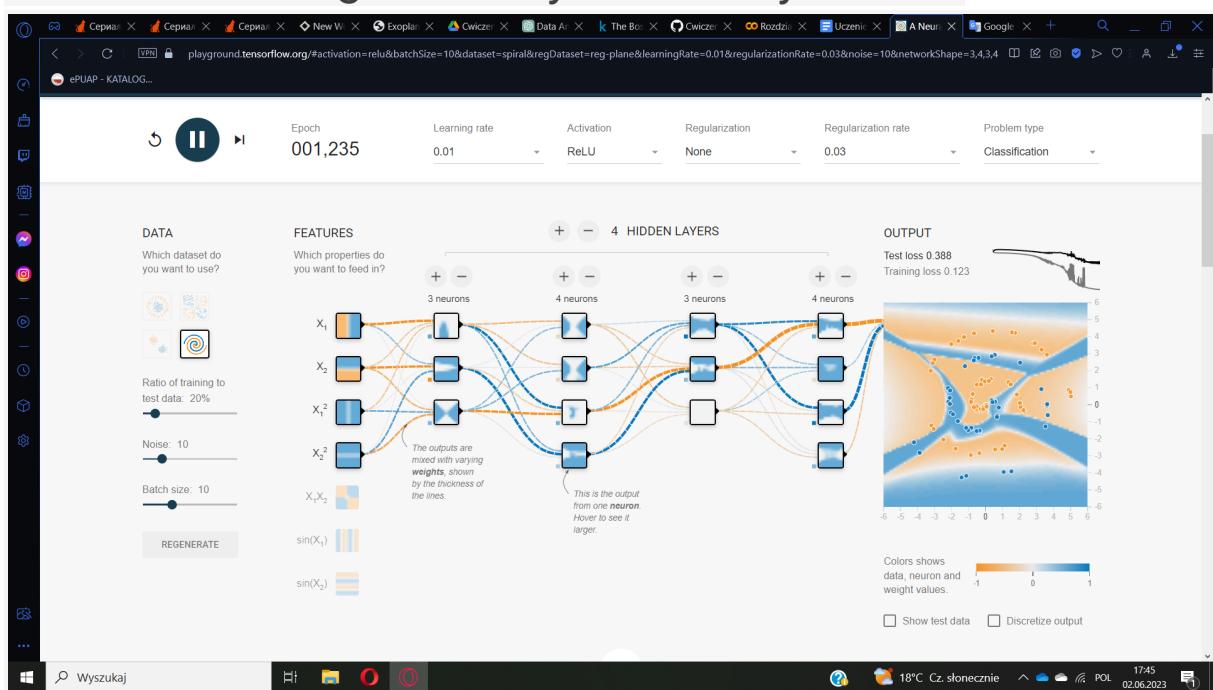


Noise: 15

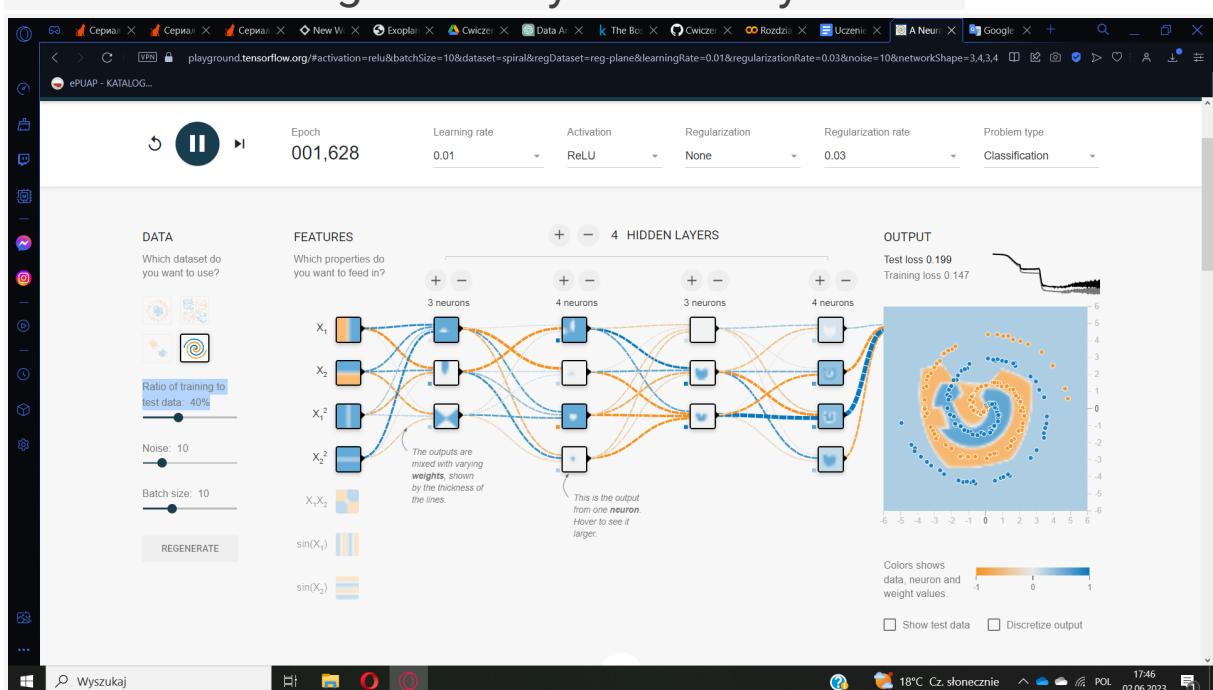


5. Sprawdzić wpływ wielkości testowego zbioru.

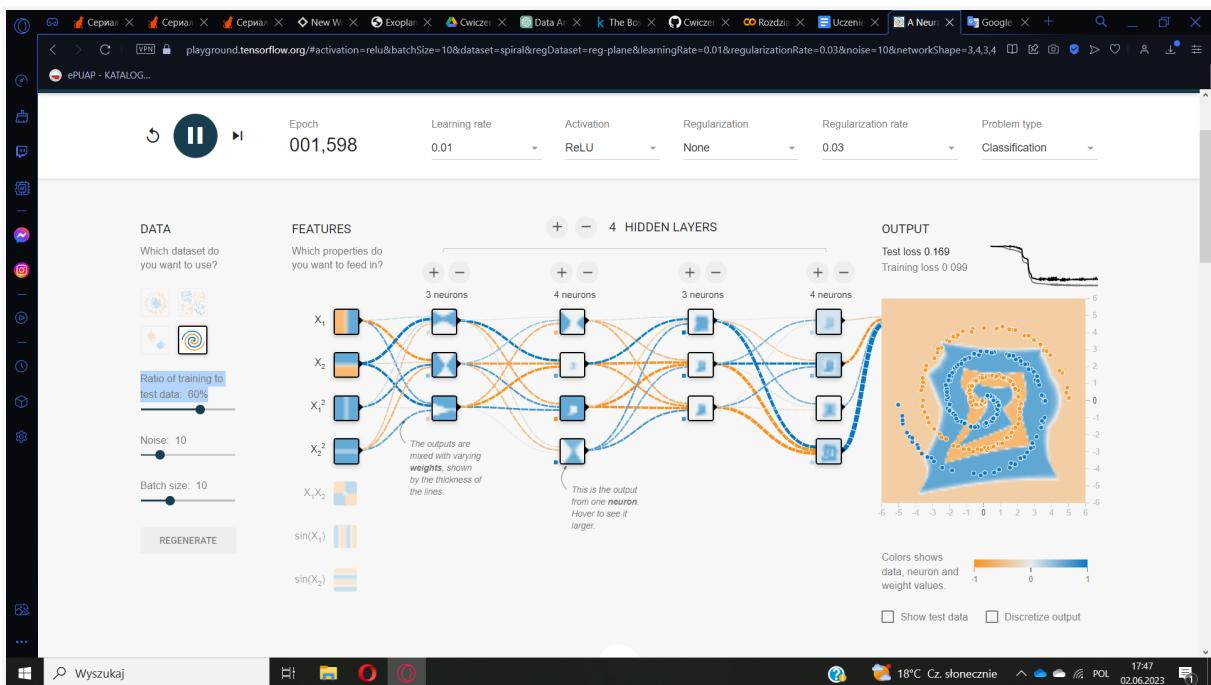
Stosunek treningu do danych testowych: 20%



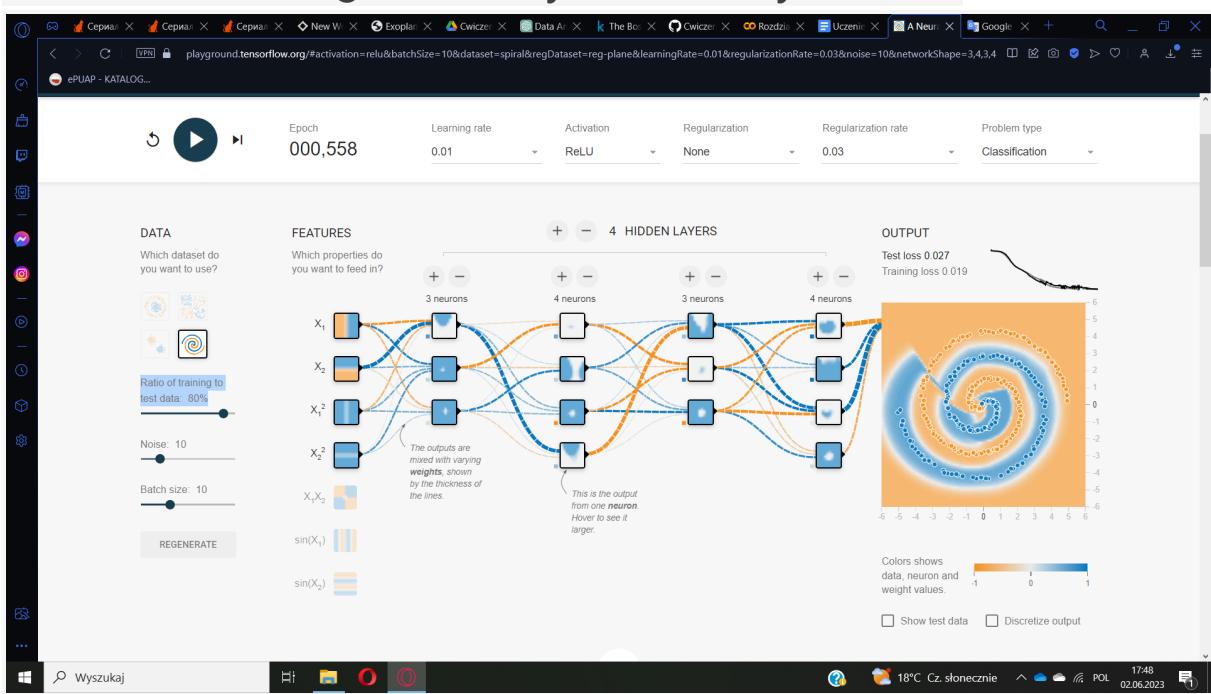
Stosunek treningu do danych testowych: 40%



Stosunek treningu do danych testowych: 60%



Stosunek treningu do danych testowych: 80%



6. Wyjaśnić pojęcia learning rate, regularization oraz regularization rate.

Learning Rate:

Learning rate (współczynnik uczenia) jest hiperparametrem, który kontroluje jak bardzo wagi modelu są aktualizowane w trakcie procesu uczenia. Określa krok, o jaki zostaną zmienione wagi na podstawie gradientu funkcji straty. Zbyt duży learning rate może prowadzić do rozbieżności modelu, podczas gdy zbyt mały może powodować zbyt wolną konwergencję lub utknienie w minimach lokalnych. Odpowiednie dobranie learning rate jest kluczowe dla skutecznego uczenia modelu.

Regularization:

Regularization (regularyzacja) to technika używana do kontrolowania przeuczenia (overfittingu) modelu. Przeuczenie występuje, gdy model bardzo dobrze dopasowuje się do danych treningowych, ale słabo generalizuje do nowych danych. Regularization pomaga zapobiegać przeuczeniu przez wprowadzenie dodatkowych ograniczeń na wagę modelu. Działa to poprzez dodanie do funkcji straty pewnego rodzaju dodatkowej penalizacji za duże wag. Popularne techniki regularizacji to L1-regularization (Lasso), L2-regularization (Ridge), ElasticNet itp.

Regularization Rate:

Regularization rate (współczynnik regularizacji) to hiperparametr, który kontroluje siłę regularyzacji. Określa, jak bardzo będą karane duże wartości wag modelu. Wyższe wartości regularyzacji prowadzą do bardziej ograniczonych wag, co może pomóc w zapobieganiu przeuczeniu. Niska wartość regularyzacji może prowadzić do mniej skomplikowanego modelu, ale może być bardziej podatna na przeuczenie. Wybór optymalnej wartości regularyzacji zależy od specyfiki problemu i struktury danych.

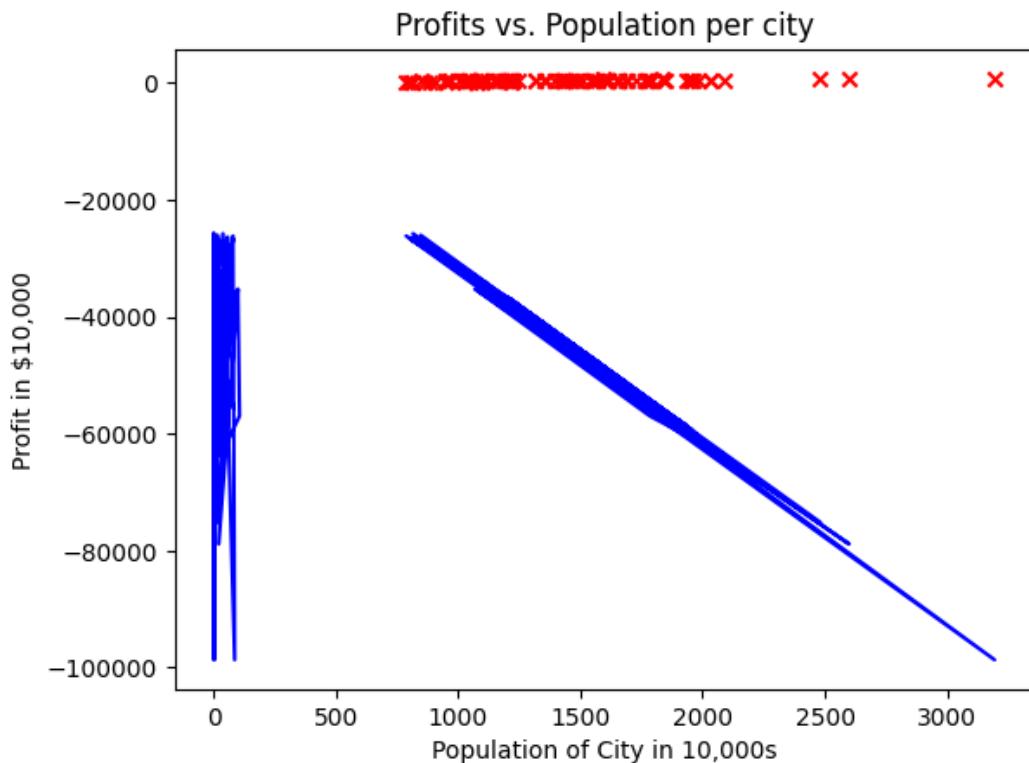
Rozdział 2 – Regresja Liniowa

Na podstawie

https://github.com/jxareas/Machine-Learning-Notebooks/blob/master/1_Supervised_Machine_Learning/Week%202.%20Regression%20with%20multiple%20input%20variables/C1_W2_Linear_Regression.ipynb

1. compute_cost()
2. compute_gradient()





Link do gist na github:

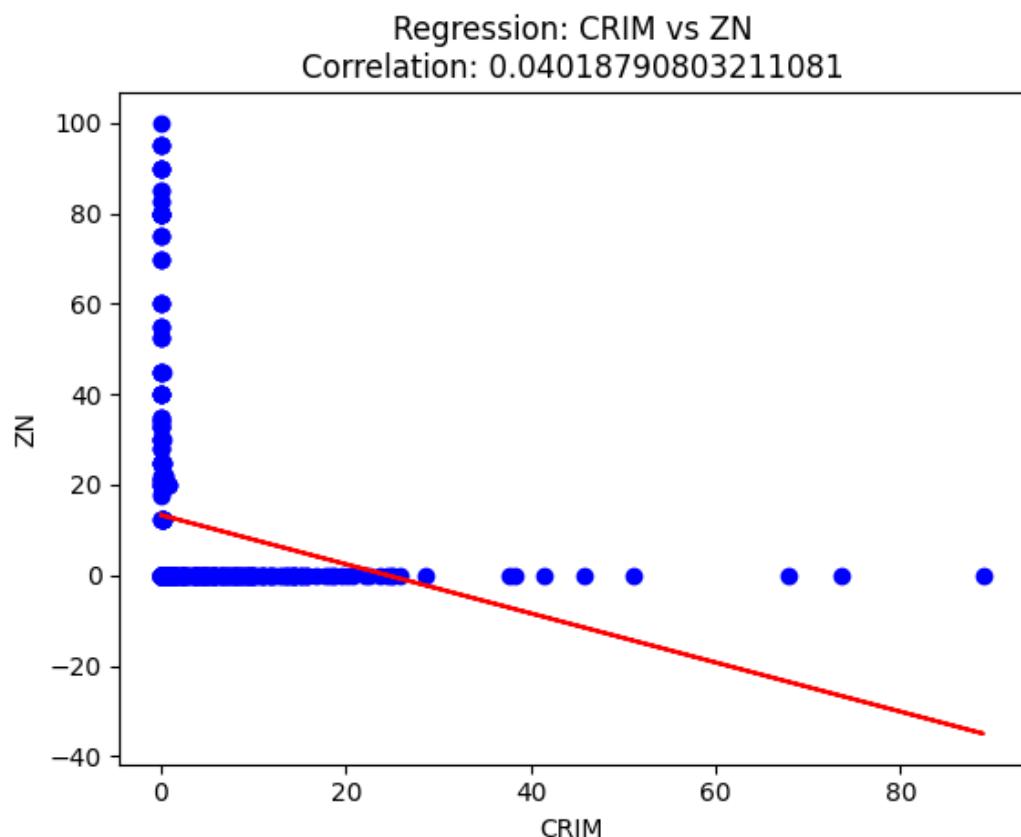
https://github.com/anksunamoonal/Uczenie_maszynowe_Cwiczenie1/blob/main/Rozdział_2.ipynb

2.2 Regresja liniowa dla pliku csv

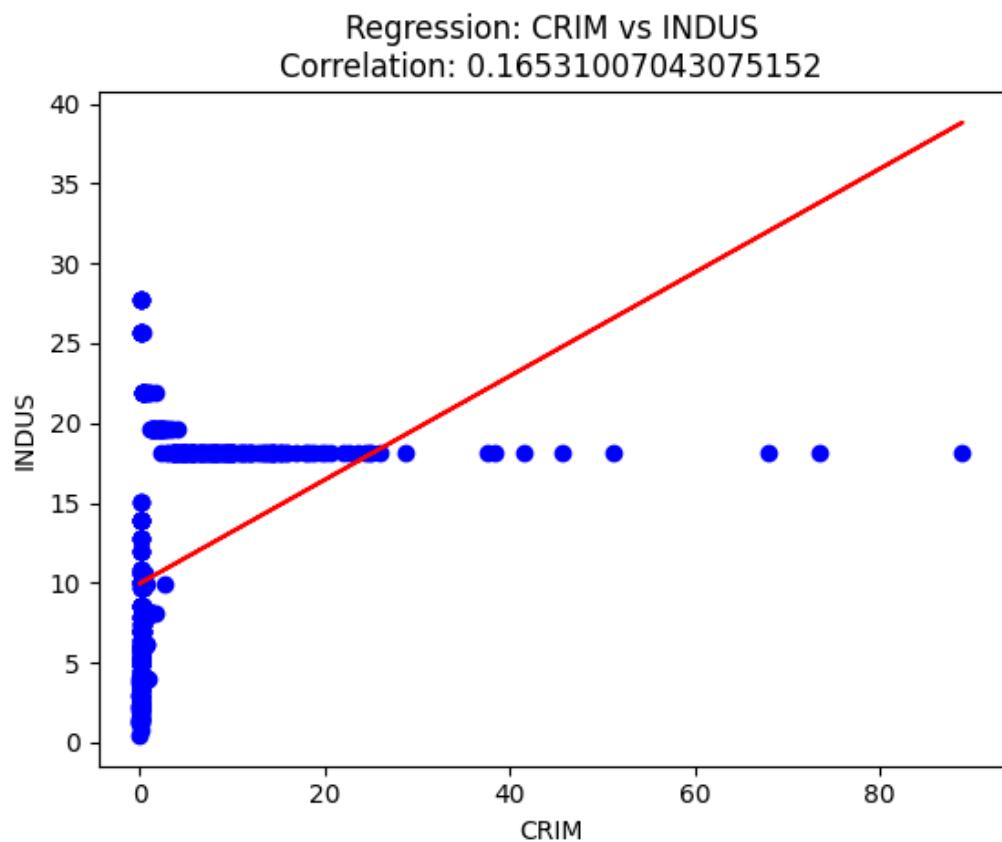
Link do gist na github:

https://github.com/anksunamoonal/Uczenie_maszynowe_Cwiczenie1/blob/main/Rozdział_2_2.ipynb

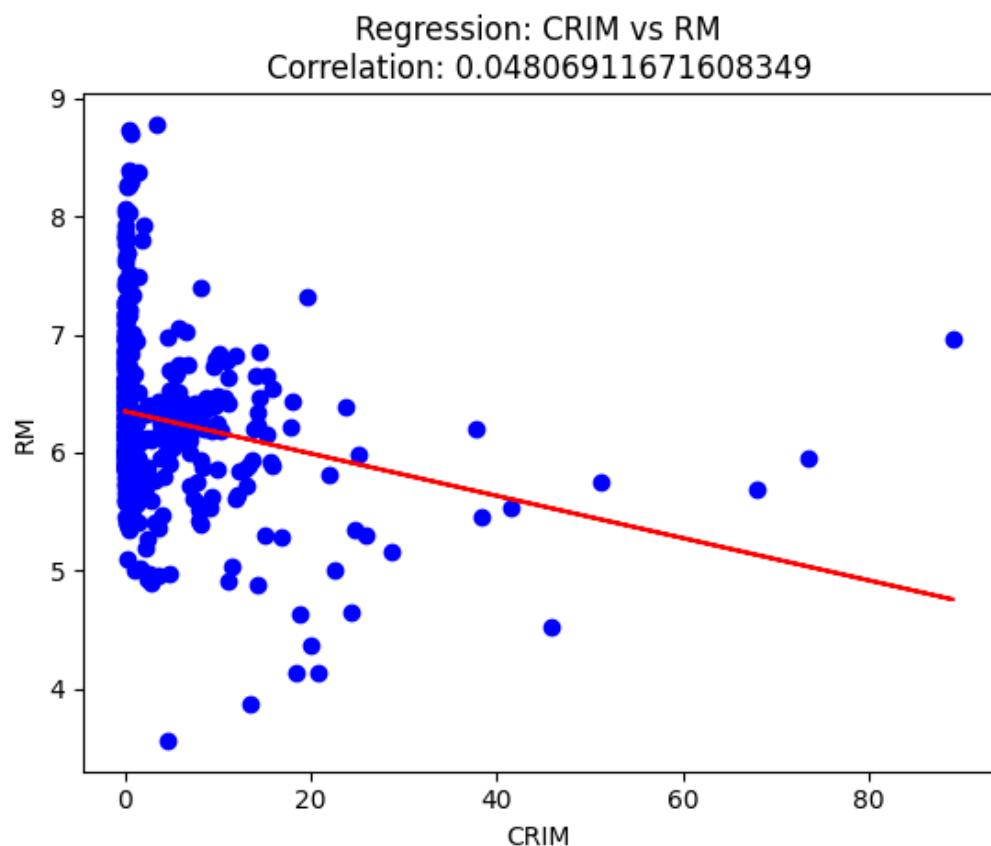
1. CRIM, ZN



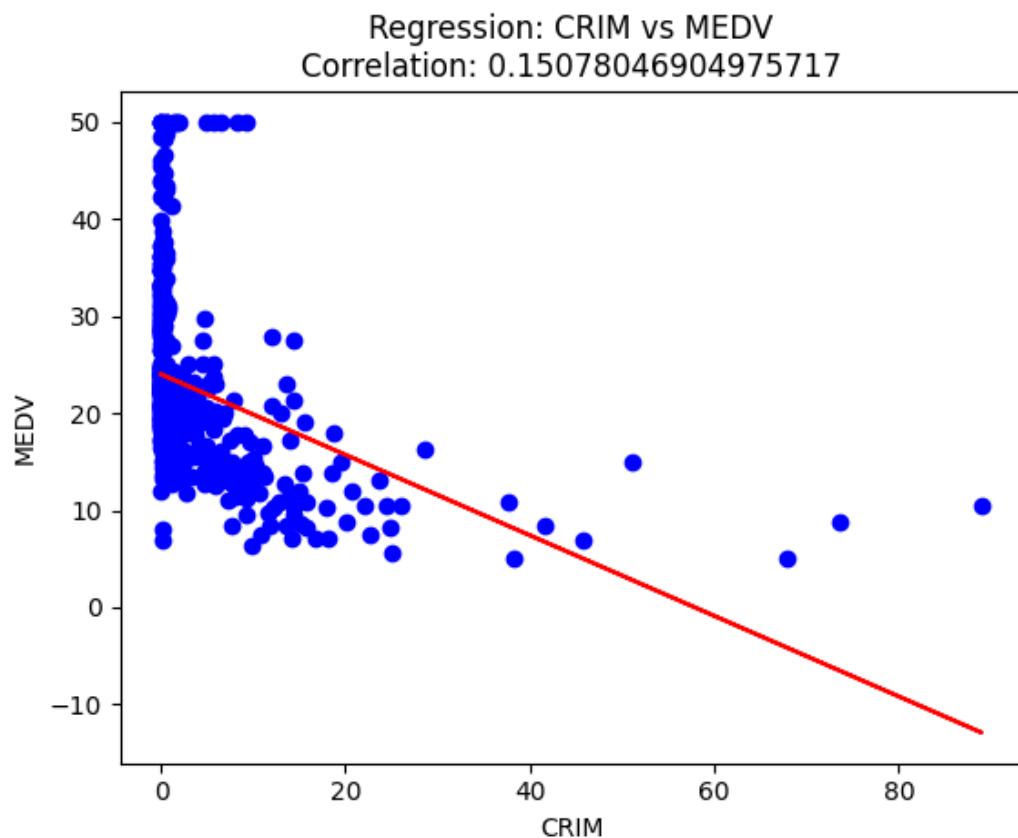
2. CRIM, INDUS



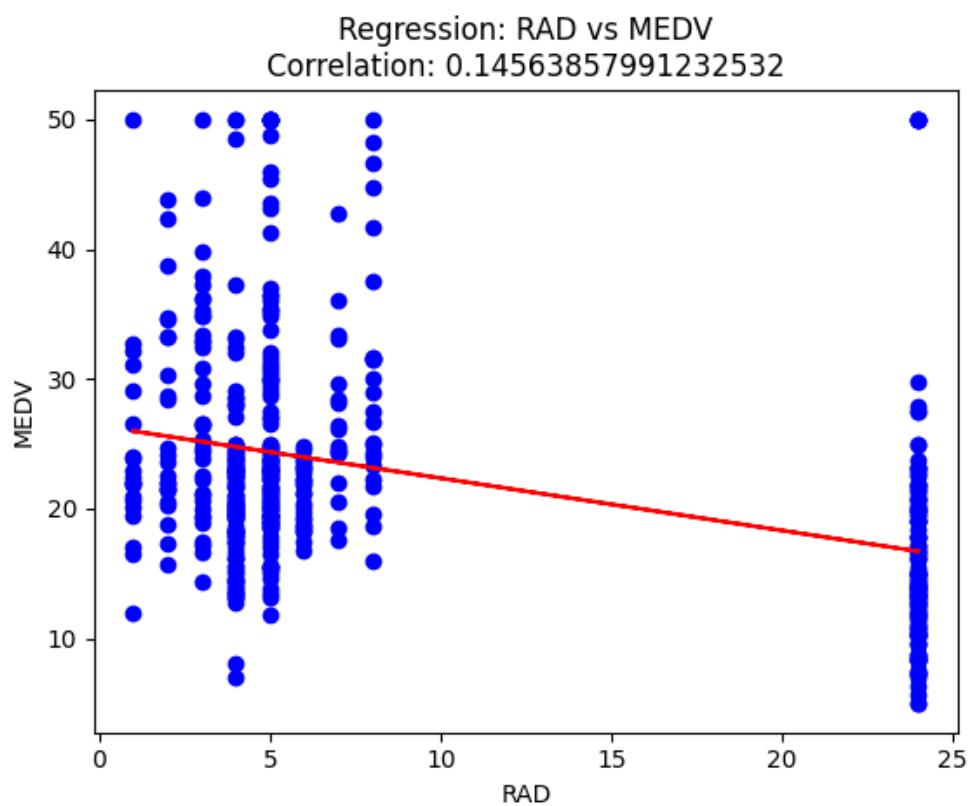
3. CRIM, RM



5. CRIM, MEDV

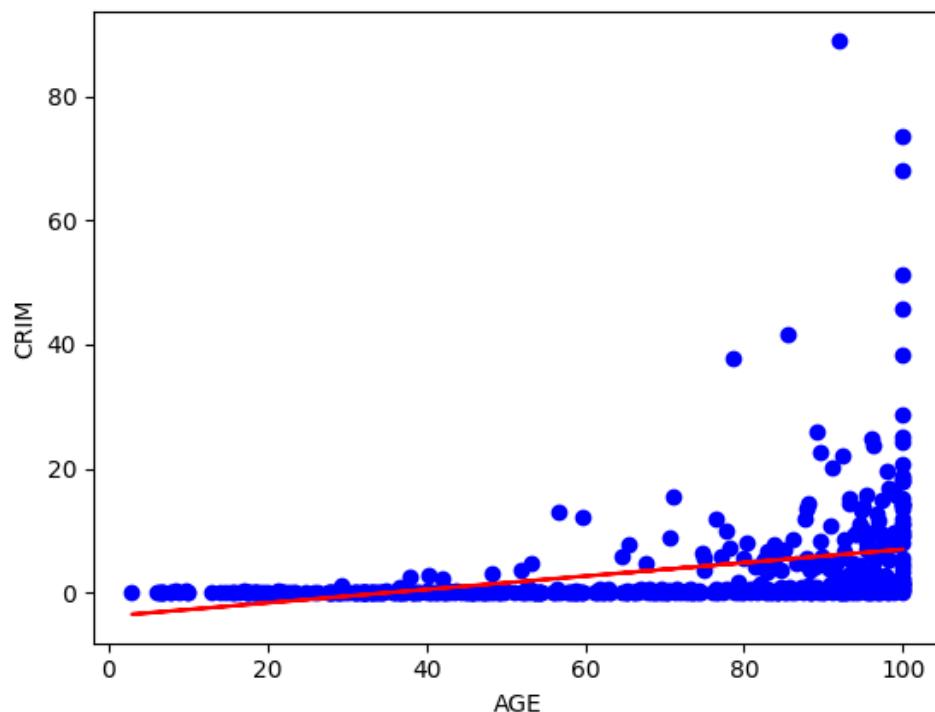


6. RAD, MEDV



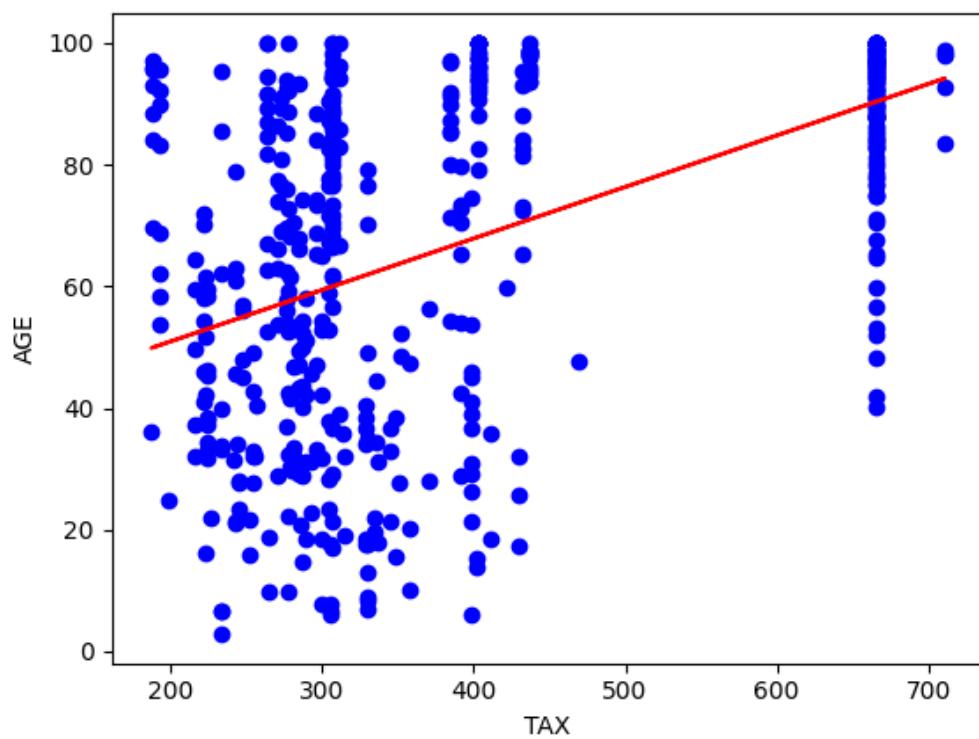
7. AGE, CRIM

Regression: AGE vs CRIM
Correlation: 0.12442145175894637

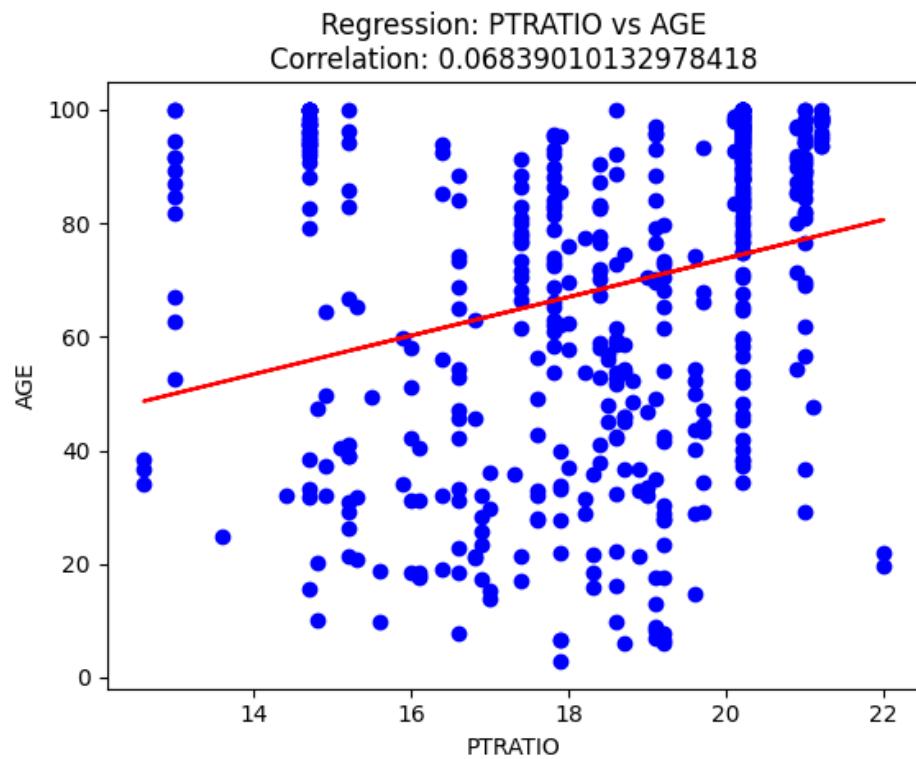


8. TAX, AGE

Regression: TAX vs AGE
Correlation: 0.2564972682387977



9. PTRATIO, AGE



Rozdział 3 – Modelowanie Funkcji Logicznych

Na podstawie <https://colab.google.com> lub lokalnej instalacji TensorFlow 2.0 wykonać poniższy kod w notatniku Jupyter:

```
from keras.models import Sequential
from keras.layers import *
import numpy as np

m = Sequential()
m.add(Dense(2,input_dim=2, activation='tanh'))
m.add(Dense(12,input_dim=2, activation='tanh'))
#m.add(Activation('tanh'))

m.add(Dense(1,activation='sigmoid'))

#m.add(Activation('sigmoid'))

X = np.array([[0,0],[0,1],[1,0],[1,1]],'float32')
Y = np.array([[0],[1],[1],[0]],'float32')

m.compile(optimizer='adam',loss='binary_crossentropy')
m.fit(X,Y,batch_size=1,epochs=200,verbose=0)
print(m.predict(X))
```

Zadania do wykonania:

1. Podać kod źródłowy dla funkcji logicznej OR, NOR, AND, NAND.
Link do gist na github:

https://github.com/anksunamoonal/Uczenie_maszynowe_Cwiczenie1/blob/main/Rozdział_3.ipynb

FUNKCJA LOGICZNA OR

```
# Funkcja logiczna OR
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], 'float32')
Y_or = np.array([[0], [1], [1], [1]], 'float32')

model_or = Sequential()
model_or.add(Dense(2, input_dim=2, activation='tanh'))
model_or.add(Dense(12, activation='tanh'))
model_or.add(Dense(1, activation='sigmoid'))

model_or.compile(optimizer='adam', loss='binary_crossentropy')
```

```
model_or.fit(X_or, Y_or, batch_size=1, epochs=200, verbose=0)
print("OR Gate Output:")
print(model_or.predict(X_or))
```

FUNKCJA LOGICZNA NOR

```
# Funkcja logiczna NOR
X_nor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], 'float32')
Y_nor = np.array([[1], [0], [0], [0]], 'float32')

model_nor = Sequential()
model_nor.add(Dense(2, input_dim=2, activation='tanh'))
model_nor.add(Dense(12, activation='tanh'))
model_nor.add(Dense(1, activation='sigmoid'))

model_nor.compile(optimizer='adam', loss='binary_crossentropy')
model_nor.fit(X_nor, Y_nor, batch_size=1, epochs=200, verbose=0)
print("NOR Gate Output:")
print(model_nor.predict(X_nor))
```

FUNKCJA LOGICZNA AND

```
# Funkcja logiczna AND
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], 'float32')
Y_and = np.array([[0], [0], [0], [1]], 'float32')

model_and = Sequential()
model_and.add(Dense(2, input_dim=2, activation='tanh'))
model_and.add(Dense(12, activation='tanh'))
model_and.add(Dense(1, activation='sigmoid'))

model_and.compile(optimizer='adam', loss='binary_crossentropy')
model_and.fit(X_and, Y_and, batch_size=1, epochs=200, verbose=0)
print("AND Gate Output:")
print(model_and.predict(X_and))
```

FUNKCJA LOGICZNA NAND

```
# Funkcja logiczna NAND
X_nand = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], 'float32')
Y_nand = np.array([[1], [1], [1], [0]], 'float32')

model_nand = Sequential()
model_nand.add(Dense(2, input_dim=2, activation='tanh'))
model_nand.add(Dense(12, activation='tanh'))
model_nand.add(Dense(1, activation='sigmoid'))

model_nand.compile(optimizer='adam', loss='binary_crossentropy')
```

```
model_nand.fit(X_nand, Y_nand, batch_size=1, epochs=200, verbose=0)
print("NAND Gate Output:")
print(model_nand.predict(X_nand))
```

2. Wypróbować dwie różne architektury (zmienić liczbę warstw ukrytych oraz ilość neuronów w tych warstwach).

Link do gist na github:

https://github.com/anksunamoonal/Uczenie_maszynowe_Cwiczenie1/blob/main/Rozdział_3_2.ipynb

I. Architektura

Poniższa Architektura posiada dwie warstwy ukryte o rozmiarach (ilość neuronów) 8 i 4

```
# Funkcja logiczna OR
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], 'float32')
Y_or = np.array([[0], [1], [1], [1]], 'float32')

model_or = Sequential()
model_or.add(Dense(16, input_dim=2, activation='tanh'))
model_or.add(Dense(8, activation='tanh'))
model_or.add(Dense(4, activation='tanh'))
model_or.add(Dense(1, activation='sigmoid'))

model_or.compile(optimizer='adam', loss='binary_crossentropy')
model_or.fit(X_or, Y_or, batch_size=1, epochs=200, verbose=0)
print("OR Gate Output:")
print(model_or.predict(X_or))
```

II. Architektura

Poniższa Architektura posiada trzy warstwy ukryte o rozmiarach (ilość neuronów) 32, 16 i 8

```
from keras.models import Sequential
from keras.layers import *
import numpy as np

# Funkcja logiczna OR
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], 'float32')
Y_or = np.array([[0], [1], [1], [1]], 'float32')

model_or = Sequential()
model_or.add(Dense(64, input_dim=2, activation='tanh'))
```

```
model_or.add(Dense(32, activation='tanh'))
model_or.add(Dense(16, activation='tanh'))
model_or.add(Dense(8, activation='tanh'))
model_or.add(Dense(1, activation='sigmoid'))

model_or.compile(optimizer='adam', loss='binary_crossentropy')
model_or.fit(X_or, Y_or, batch_size=1, epochs=200, verbose=0)
print("OR Gate Output:")
print(model_or.predict(X_or))
```

3. Odpowiedzieć na pytania:

- a) Jaki wpływ na szybkość trenowania ma architektura sieci.

Dodawanie większej liczby warstw ukrytych może zwiększyć złożoność modelu, co może prowadzić do dłuższego czasu trenowania. Większa liczba neuronów w warstwach ukrytych zwiększa liczbę parametrów w modelu, co może skutkować dłuższym czasem trenowania, ponieważ model musi nauczyć się optymalnych wartości wag dla większej liczby neuronów.

- b) Czy wynik końcowy jest satysfakcyjny?

Ostateczna ocena zadowolenia z wyniku zależy od oczekiwanych rezultatów. W przypadku funkcji logicznej OR, można by oczekwać wyższej wartości dla przypadków [0, 1], [1, 0] i [1, 1], a niższej wartości dla przypadku [0, 0].

- c) Dlaczego nie można osiągnąć idealnego dopasowania wyników przewidywanych do wyników wzorcowych?

Na wynik który otrzymujemy pod koniec trenowania ma wpływ wiele czynników, przykładowo szum, który może utrudnić modelowi osiągnięcie idealnego dopasowania, ponieważ nie ma jednoznacznej zależności między wejściem a oczekiwany wynikiem.

- d) Jaki wpływ na trenowanie ma wybór innego optymalizatora.

Sprawdzając zmienioną architekturę OR z różnymi optymalizatorami, wyniki są drażniąco różne. Najbardziej zbliżone do oczekiwanych rezultatów otrzymujemy testując optymalizatory ADAM i NADAM. Sprawdzono sieć także z innymi optymalizatorami ADADELTA i ADAGRAD, lecz przy tych optymalizatorach nie udało się uzyskać w miarę satysfakcyjnych wyników.

Każdy optymalizator ma zalety oraz ograniczenia, które mogą wpływać na efektywność procesu uczenia. Różnice w wynikach uzyskiwanych przy użyciu różnych optymalizatorów mogą wynikać z różnych strategii aktualizacji wag i adaptacji współczynnika uczenia.