O uczeniu maszynowym i sieciach neuronowych

Paweł Rychlikowski

Instytut Informatyki UWr

7 maja 2021

Uczenie maszynowe. Przypomnienie

- Rozważaliśmy uczenie **z nadzorem** (to znaczy sytuację, gdy mamy pewne przykłady i chcemy je uogólnić)
- Rozważamy dwie ważne klasy takich zadań:
  1. Klasyfikacji (wybór klasy dla przykładu, zbiór klas jest skończony i niezbyt duży)
  2. Regresji (wybór wartości liczbowej dla przykładu)

Uczenie maszynowe. Dopasowanie wzorców

Uwaga

Dopasowanie wzorców (pattern matching)

Rozważaliśmy dwa rodzaje wzorców:

- Średnia cyfra (jeden wzorzec dla każdej cyfry)
- Każda cyfra jest wzorcem (do klasyfikacji cyfry D znajdujemy K najbliższych wzorców i przeprowadzamy głosowanie)

Uczenie funkcji liniowej

- Klasyfikacji (czy regresji) możemy dokonywać na bazie wartości:

$$\sum_{i=0}^{N} w_i \phi_i(x)$$

- Cechami (dla MNIST-a) są po prostu wartości kolejnych pikseli
- Jedna funkcja „wystarcza” dla binarnego zadania typu: czy cyfra jest piątką? (i jak bardzo)
- Do rozwiązywania MNIST-a potrzebujemy 10 takich funkcji, wybieramy cyfrę dla tej funkcji, która zwraca największą wartość.

Funkcja liniowa i Reversi

- Możemy zdefiniować zadanie uczenia (regresji) dla Reversi:

  Widząc sytuację na planszy w ruchu 20 postaraj się przewidzieć zakończenie gry.

- Cechy: binarne cechy mówiące o zajętości pola.
- Przykładowo:
  Czy pole (4,5) jest czarne?
  Czy pole (1,1) jest białe?

Uwaga

Taki mechanizm byłby użyteczną funkcją heurystyczną, do użycia np. w algorytmie MiniMax.

Funkcja kosztu

Definicja

Funkcja kosztu (loss) opisuje, jak bardzo niezadowoleni jesteśmy z działania naszego mechanizmu (klasyfikatora, przewidywacza wartości).

Funkcja kosztu jest określona na: danych uczących (x,y) oraz wagach (parametrach klasyfikatora). Przy czym dane uczące traktujemy jako parametr, a wagi – jako właściwe argumenty.

Przykładowa funkcja kosztu: błąd średniokwadratowy

Funkcja kosztu (2)

Średniokwadratowa funkcja straty

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y)\in D_{\text{train}}} (f_{\mathbf{w}}(x) - y)^2$$

Wariant liniowy:

$$f_{\mathbf{w}}(x) = \sum_{i=0}^{N} w_i \phi_i(x)$$

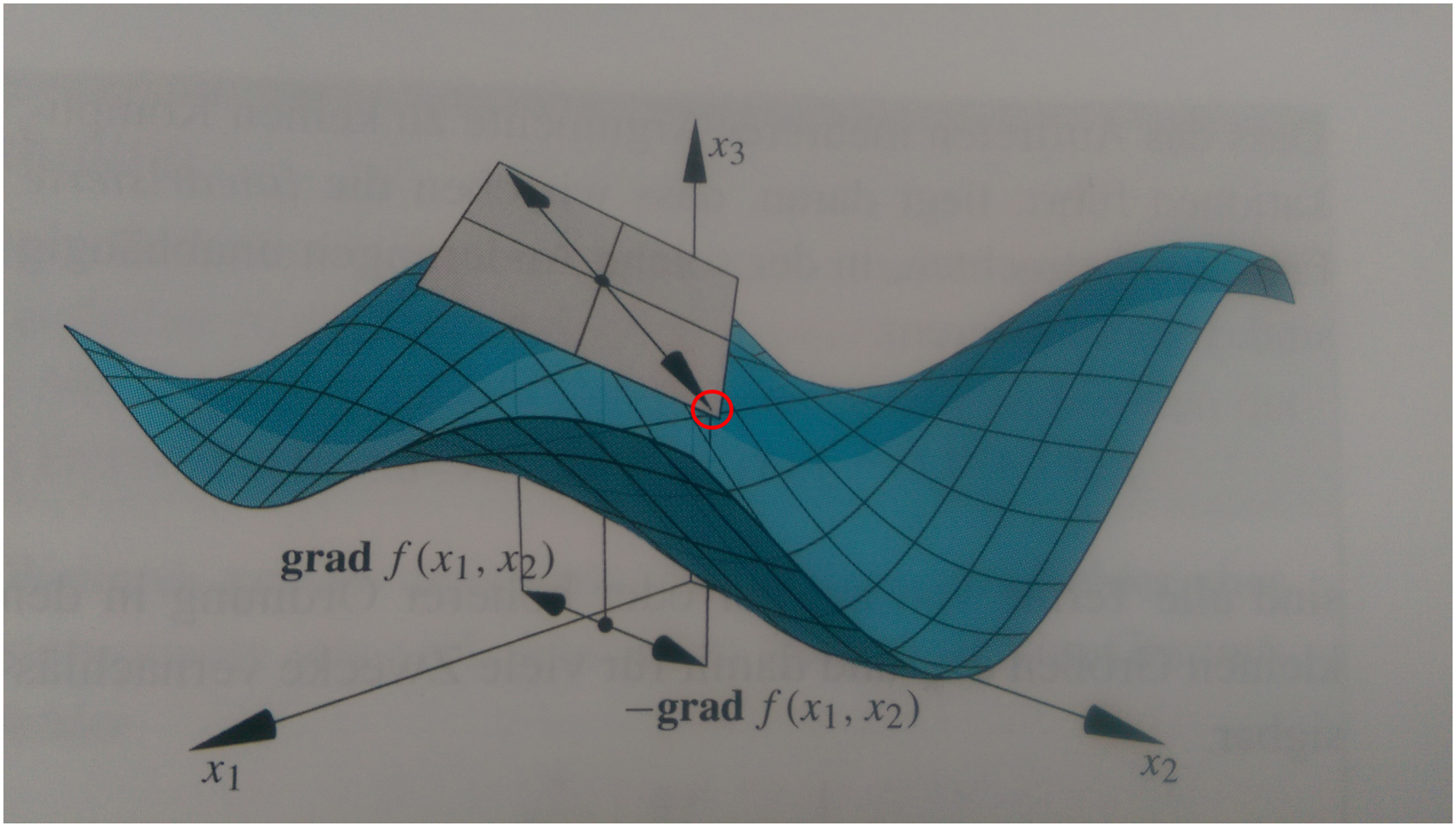
Gradient

- Gradient jest wektorem pochodnych cząstkowych.
- Wskazuje kierunek największego wzrostu funkcji.

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

Gradient

Wizualizacja gradientu w dla funkcji $\mathcal{R}^2 \rightarrow \mathcal{R}$



Jeszcze o gradiencie

Wzór

Pochodna po w_i (dla liniowej funkcji $f_{\mathbf{w}}$

$$\frac{1}{|D_{\text{train}}|} \sum_{(x,y)\in D_{\text{train}}} 2 \cdot (f_{\mathbf{w}}(x) - y) \cdot w_i \phi_i(x)$$

- Obliczenie gradientu wymaga przejścia przez cały zbiór uczący
- Maksymalizacja funkcji: dodawanie gradientu przemnożonego przez małą stałą (minimalizacja: odejmowanie)

Stochastic Gradient Descent

Obliczamy nie cały gradient, tylko jego składnik, związany z jednym egzemplarzem danych uczących i jego dodajemy (przemnożonego przez stałą)

Koniec części I

Sieci neuronowe w paru prostych slajdach

- Wybierzemy absolutne minimum tego, co należy wiedzieć o sieciach neuronowych
- Oczywiście nie będzie to w pełni kompletna wiedza.

Slajd 1. Neuron

- Neuron to funkcja $f : \mathcal{R}^n \to \mathcal{R}$

$$f(x_1 \dots x_n) = \sigma(\sum_{i=1}^{n} w_i x_i + b)$$

- σ jest jakąś ustaloną funkcją nieliniową, raczej rosnącą, raczej różniczkowalną, na przykład: $\max(0, v)$, albo $\tanh(v)$
- Wygodna (jak za chwilę zobaczymy) jest notacja wektorowo-macierzowa, w niej mamy:

$$f(\mathbf{x}) = \sigma(\mathbf{w}^T \cdot \mathbf{x} + b)$$

Slajd 2. Prosta sieć neuronowa

- Warstwa to funkcja $\mathcal{R}^n \to \mathcal{R}^m$.
- Najbardziej typowa warstwa wyraża się wzorem:

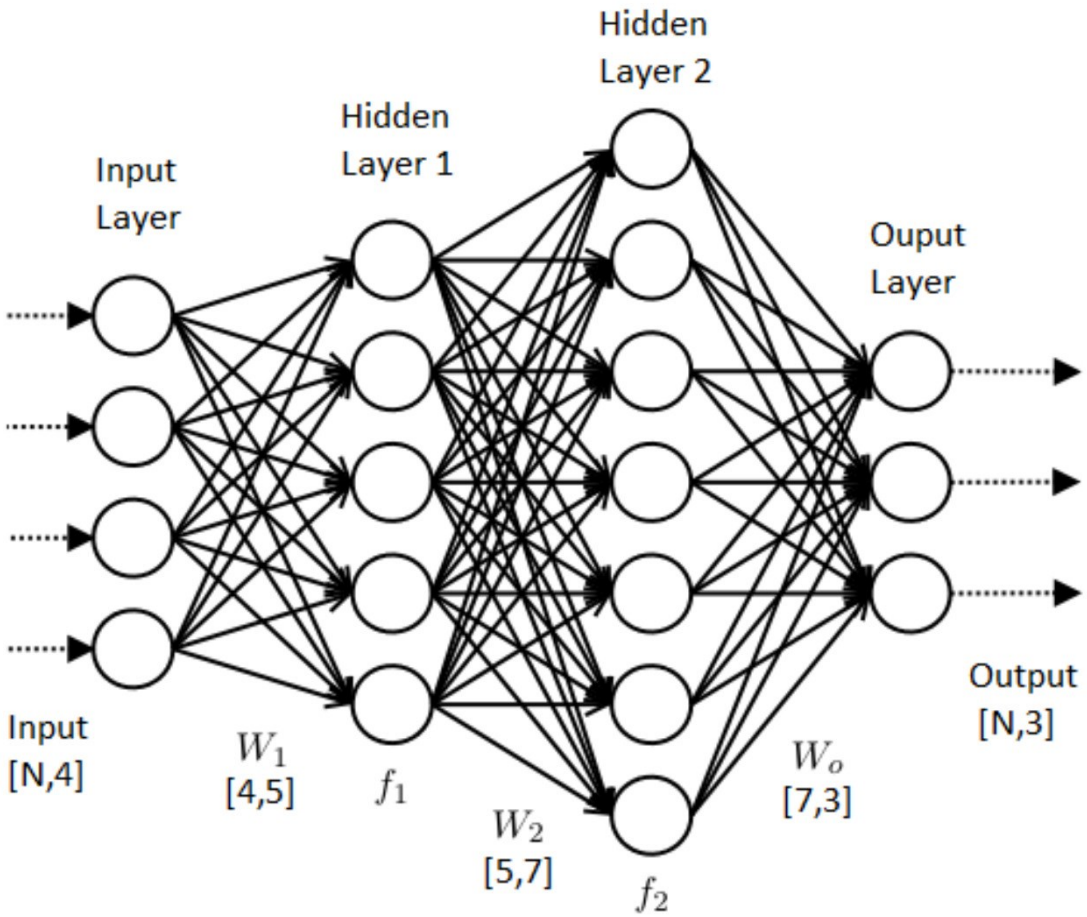
$$L(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- **Uwaga:** $\mathbf{W}$ jest macierzą wag (złożoną z wektorów wag), a $\sigma(y_1 \dots y_m) = (\sigma(y_1) \dots \sigma(y_m))$

Definicja

Sieć neuronowa typu MLP jest złożeniem warstw (z różnymi macierzami wag dla każdej warstwy).

Slajd 2b. Prosta sieć neuronowa



Źródło: VIASAT (https://medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4)

Slajd 3. Uczenie sieci

Zadanie

Danymi jest ciąg $(\mathbf{x}_i, \mathbf{y}_i)$ opisujący porządane zachowanie sieci S oraz architektura tejże sieci (liczba warstw, ich wymiary, funkcja/funkcje σ).
Chcemy tak dobrać parametry ($\mathbf{W_k}$ oraz $\mathbf{b_k}$) żeby dla każdego i

$$S(\mathbf{x}_i) \approx \mathbf{y}_i$$

Funkcja kosztu

Powyższe zadanie formalizujemy jako zadanie znalezienia takich parametrów, że koszt błędów jest jak najmniejszy. Przykładowo, jeżeli wyjściem jest liczba, to możemy wybrać:

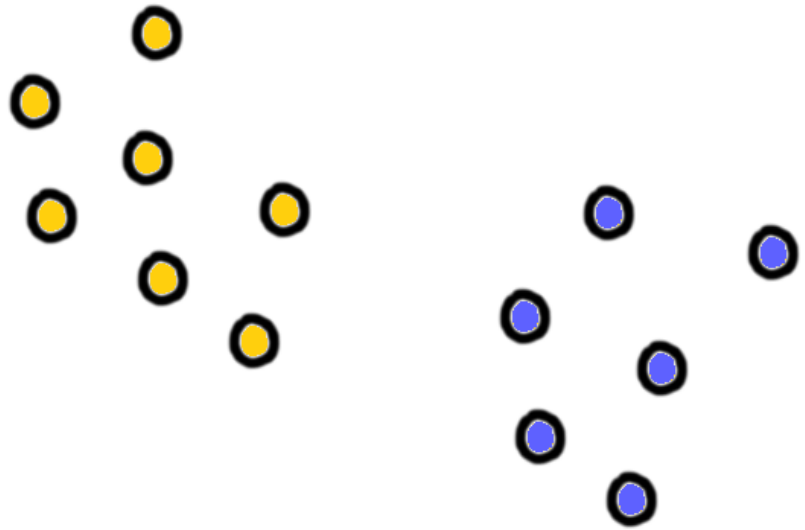
$$\text{Loss}(\theta) = \sum_{i}^{n} (S_\theta(\mathbf{x}_i) - y_i)^2$$

Sieci neuronowe w praktyce. Klasyfikacja

Ogólne założenia (przypadek dwóch klas)

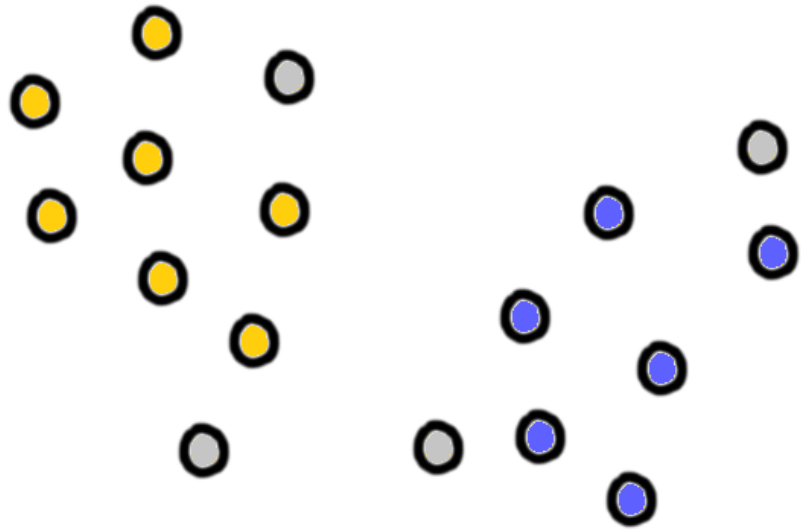
Mamy jakiś zbiór przykładów pozytywnych i negatywnych, interesuje nas mechanizm, który będzie poprawnie klasyfikował nieznane przykłady.

Klasyfikacja w $\mathcal{R}^2$



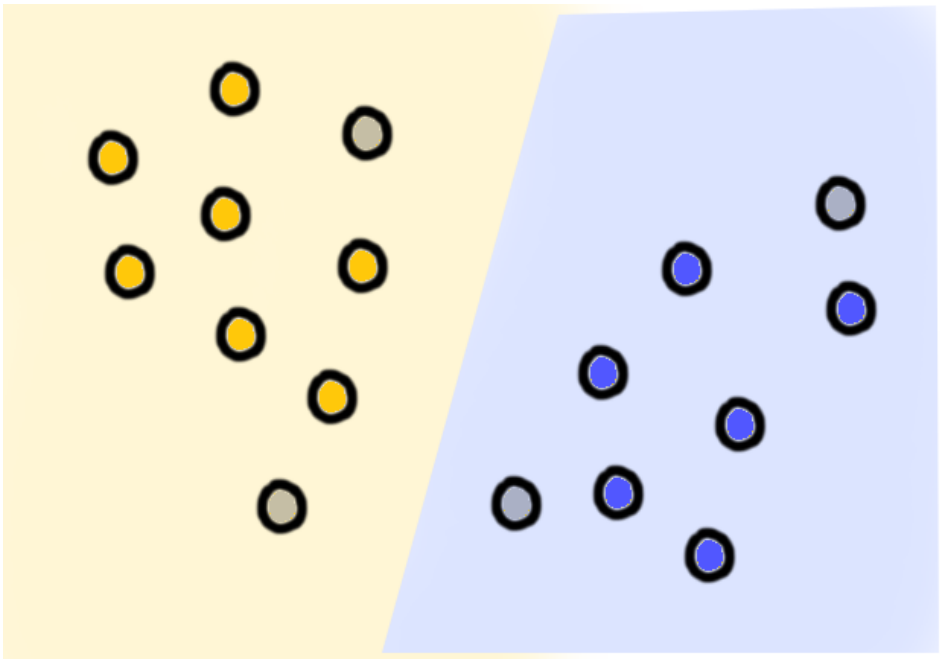
- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określać kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

Klasyfikacja w $\mathcal{R}^2$



- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określać kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

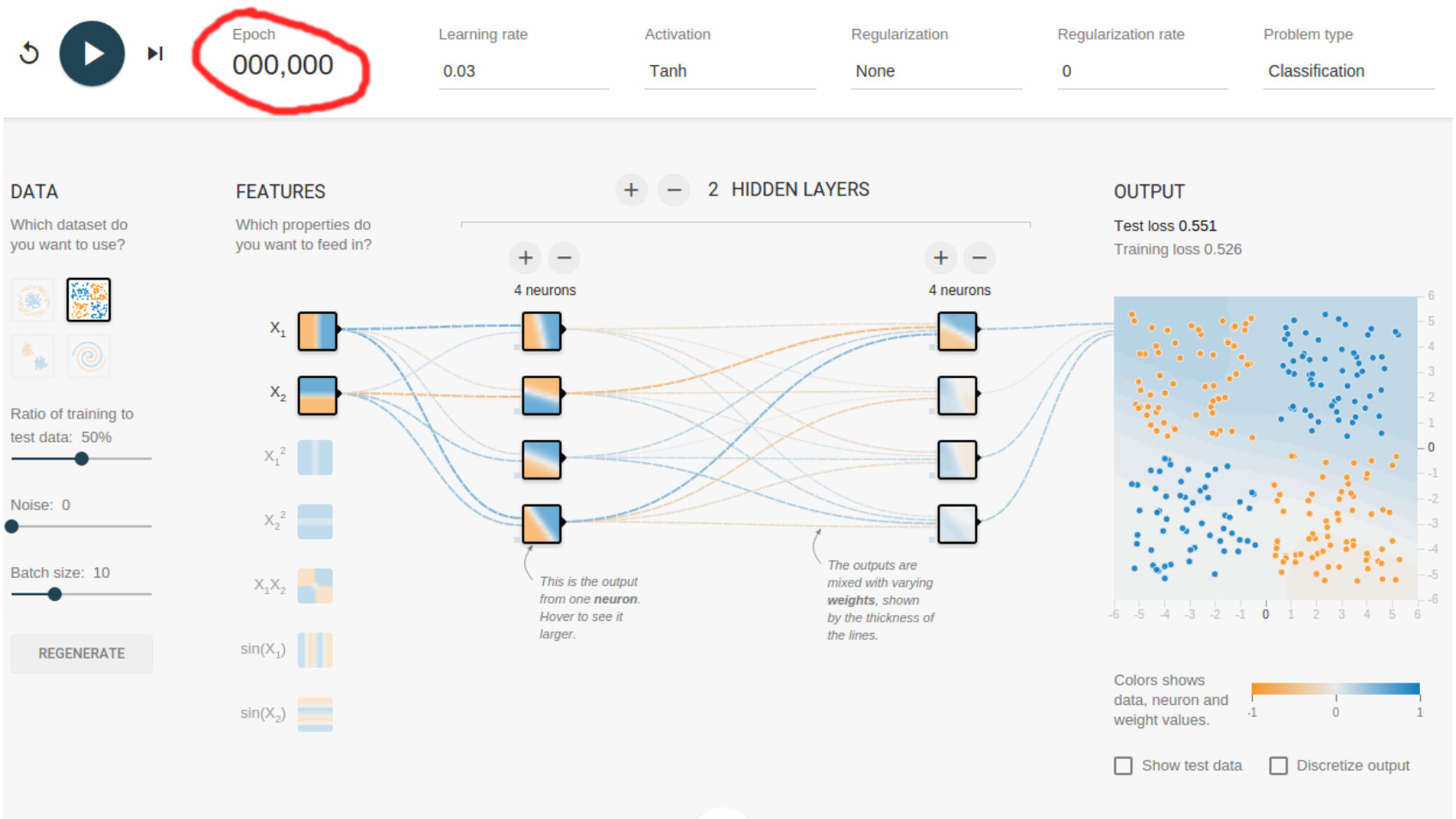
Klasyfikacja w $\mathcal{R}^2$



- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określać kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

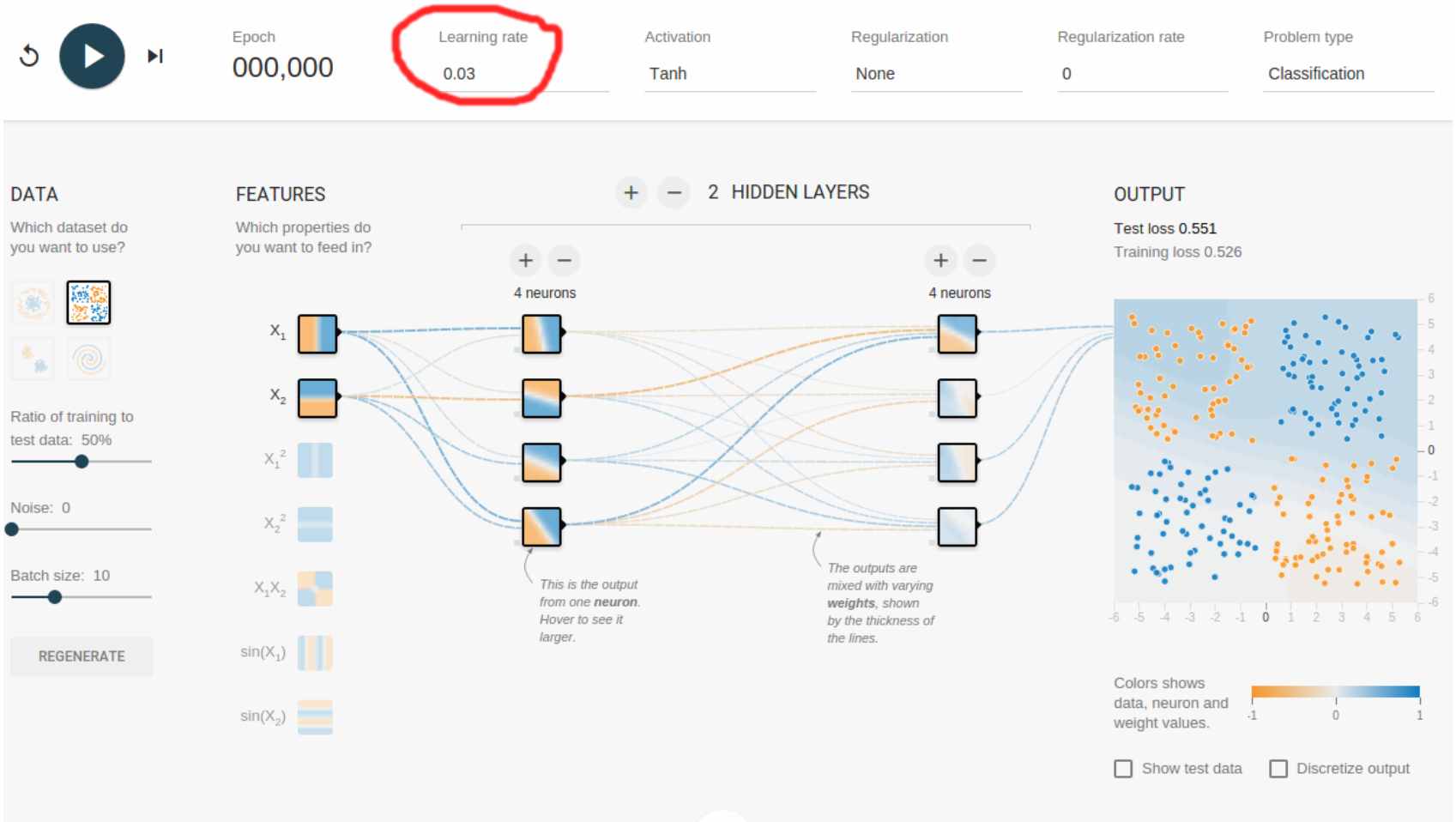
Spróbujmy poeksperymentować chwilę z Tensorflow Playground

Plac zabaw dla tensorflow. Ważne pojęcia (1)



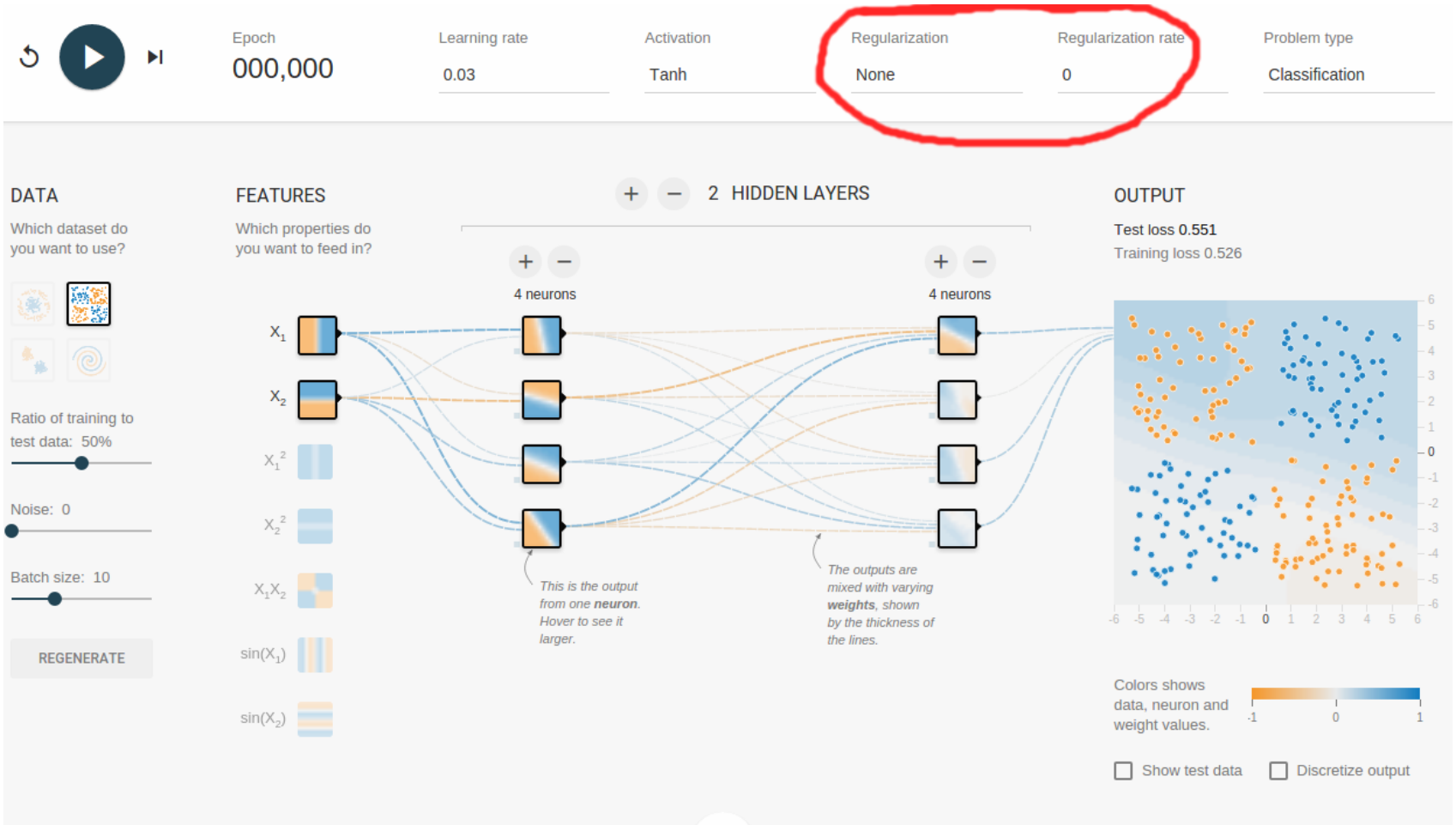
Epoka: etap uczenia, w którym uwzględnione są wszystkie dane uczące.

Plac zabaw dla tensorflow. Ważne pojęcia (2)



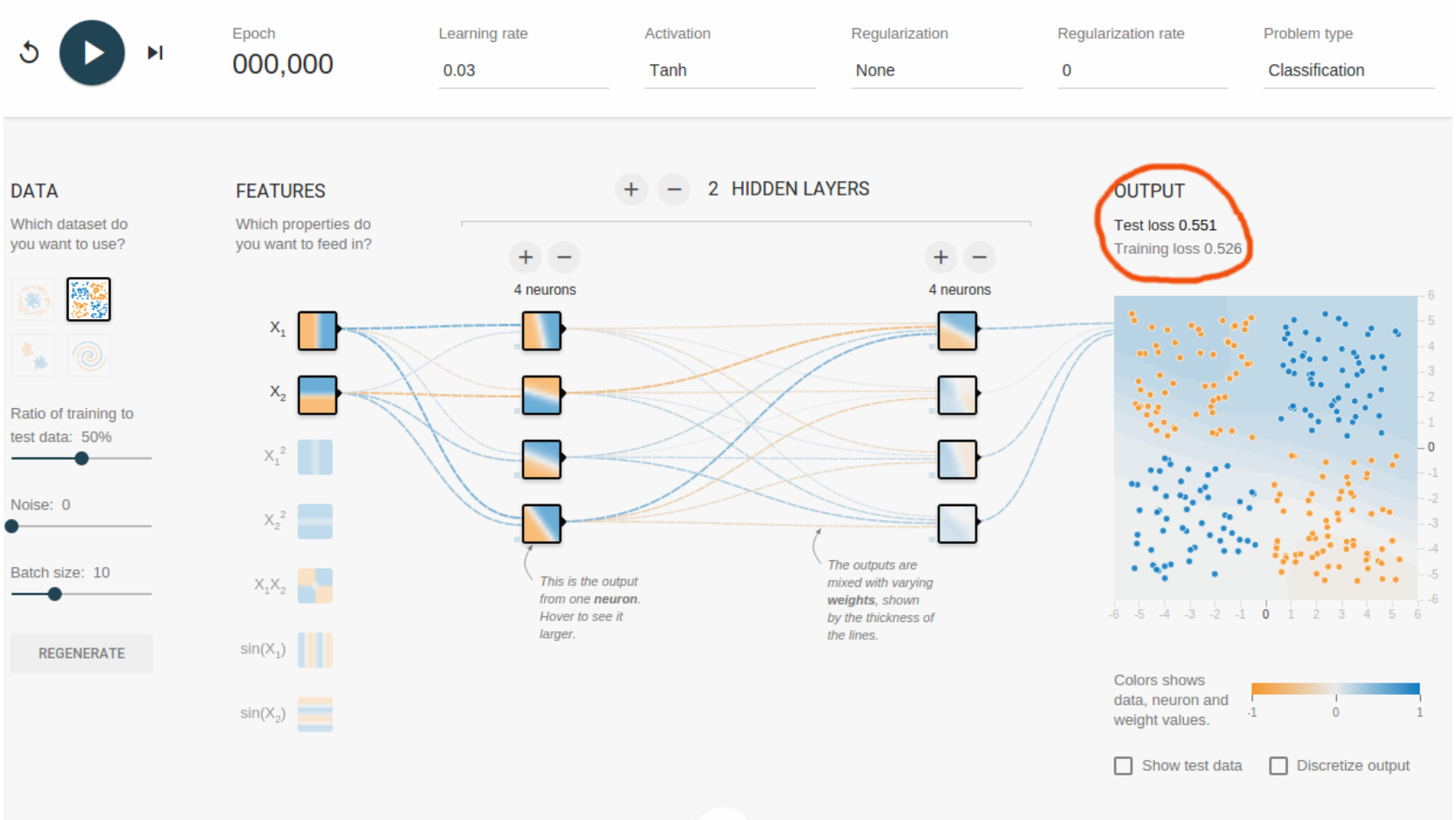
Learning rate: stała przez którą mnożone są delty wag. Za duża może dać chaotyczne zachowanie, za mała: bardzo wolny postęp.

Plac zabaw dla tensorflow. Ważne pojęcia (3)



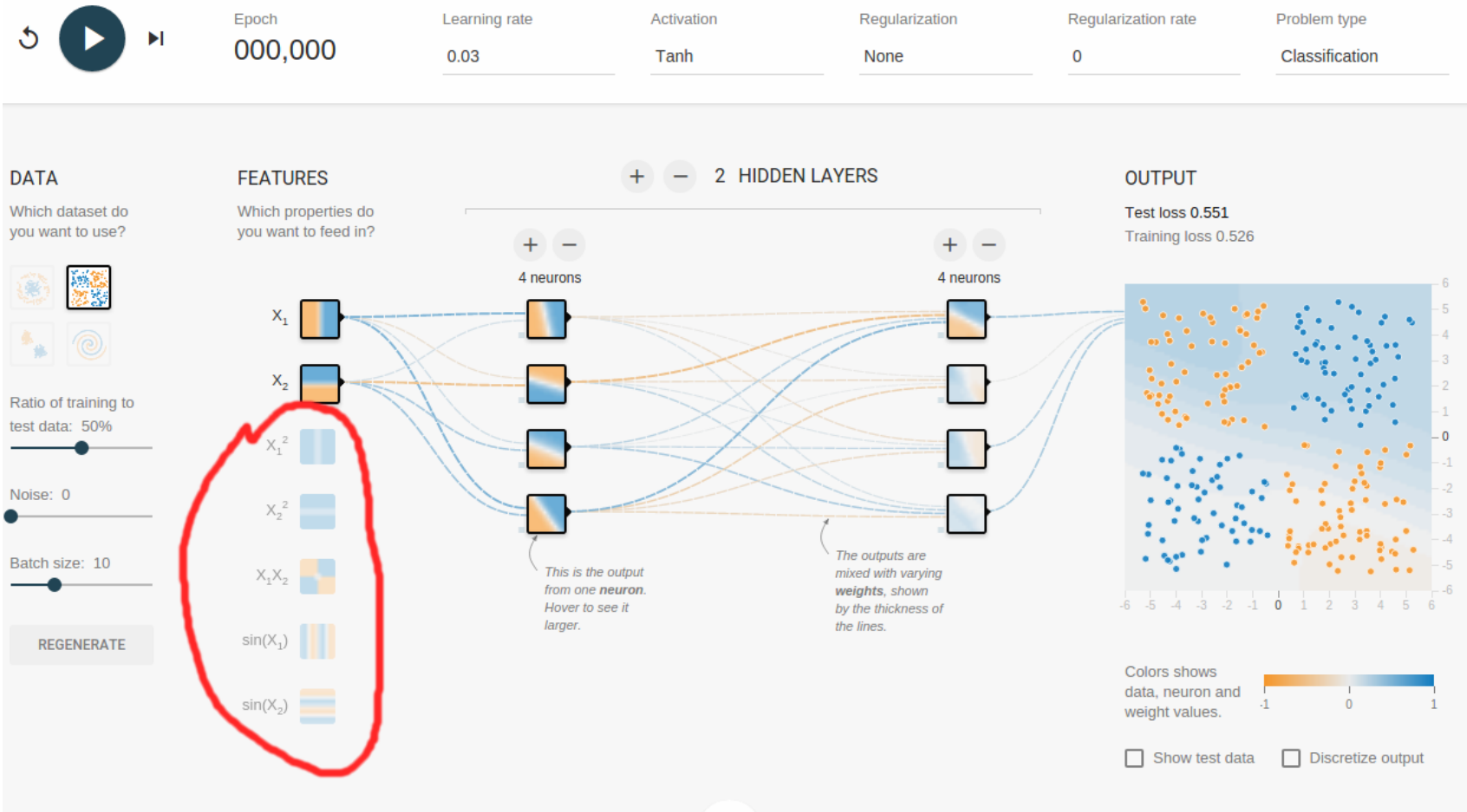
Regularyzacja: dołożenie do uczenia wymagania, by wagi nie były zbyt duże. Może dać większą stabilność uczenia (zob. tablica).

Plac zabaw dla tensorflow. Ważne pojęcia (4)



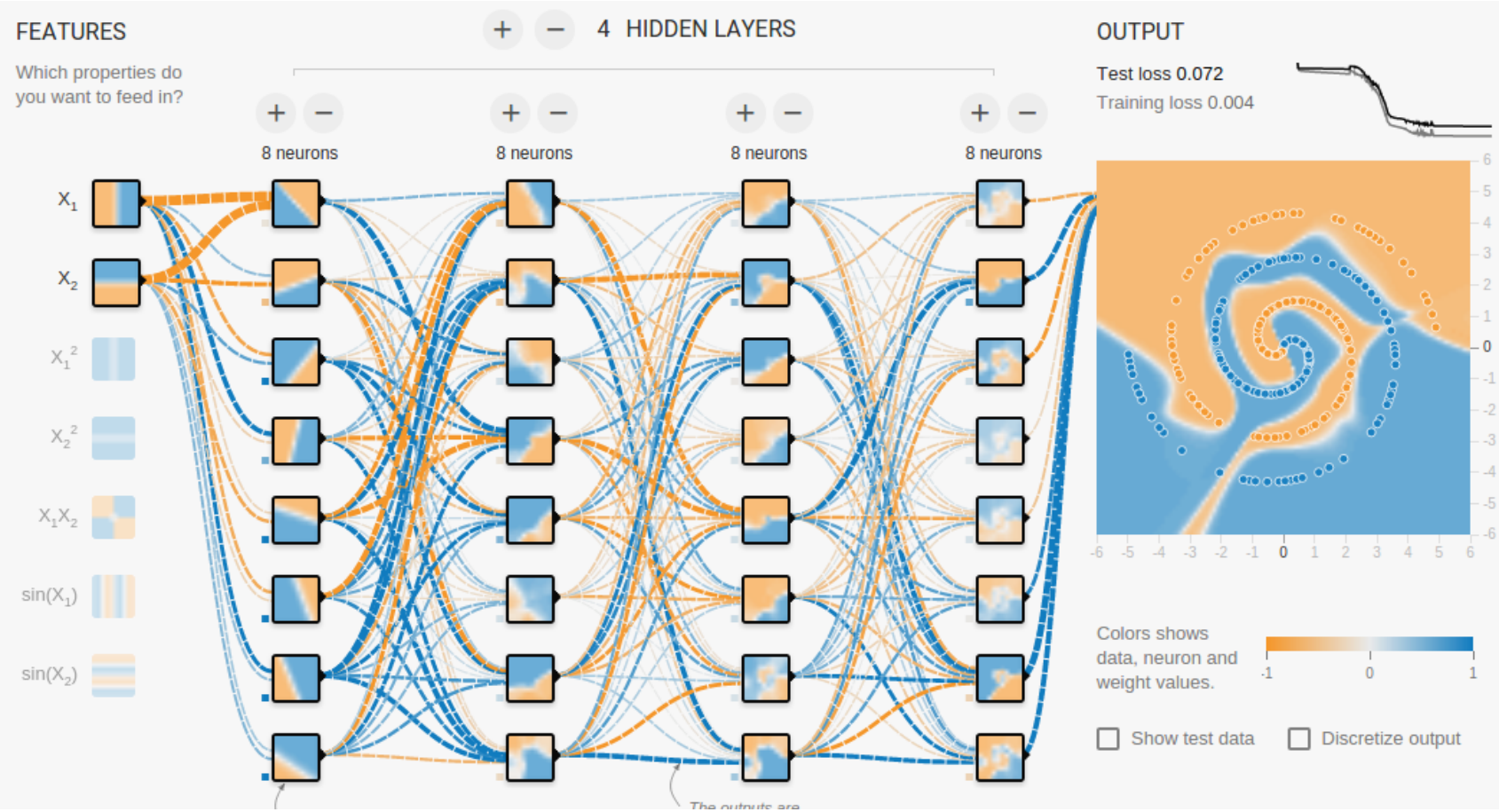
Test loss/training loss: wartość kosztu dla zbioru testowego i uczącego (oczywiście pierwsza zawsze większa).

Plac zabaw dla tensorflow. Ważne pojęcia (5)



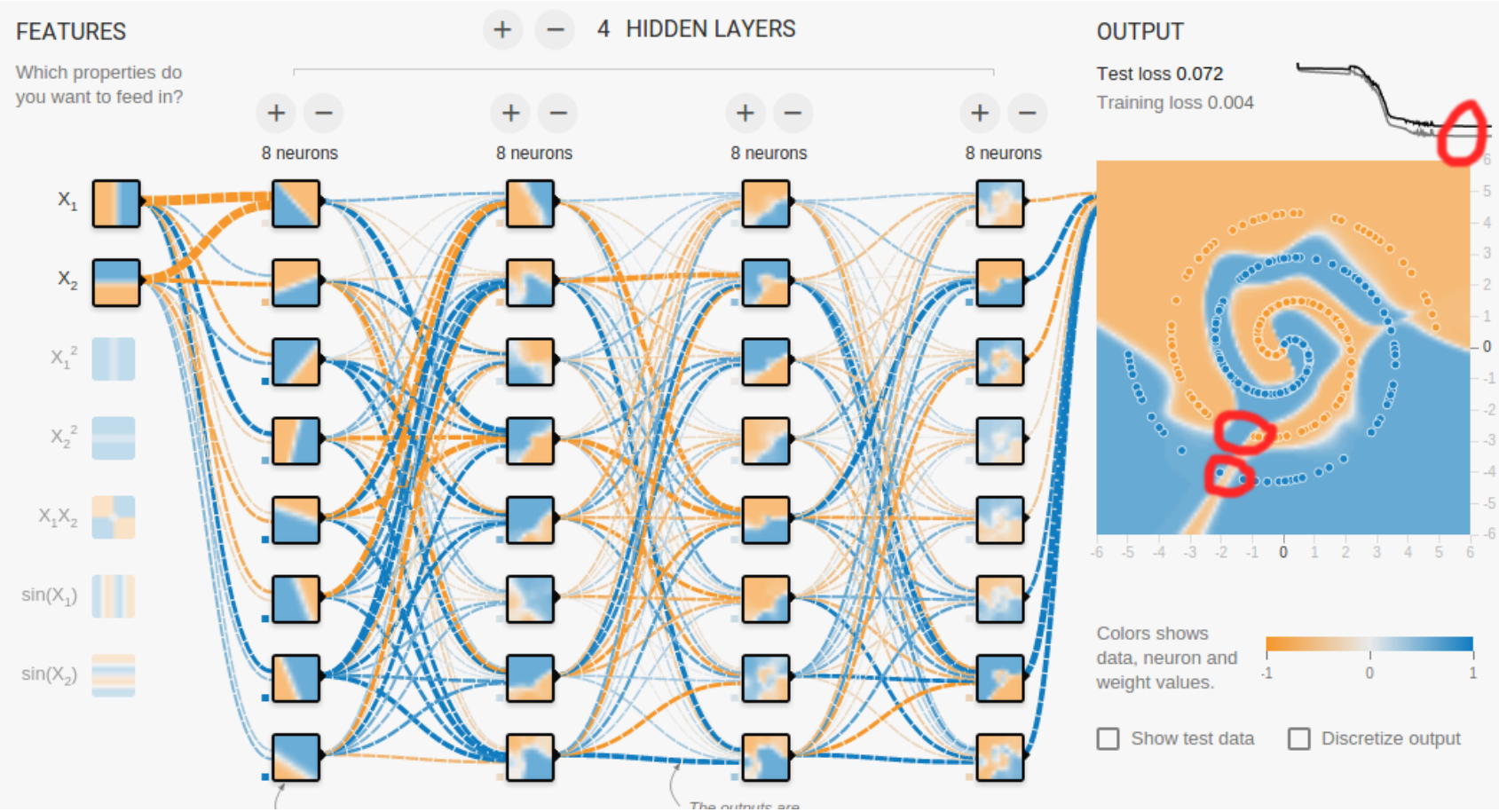
Feature engineering: proces tworzenia własnych cech dla konkretnych przypadków. Dobre cechy mają związek z zadaniem.

Plac zabaw dla tensorflow. Ważne pojęcia (6)



Przeuczenie (overfiting): sytuacja, w której sieć dostosowuje się do nieistotnych fluktuacji danych uczących, co pogarsza generalizację.

Plac zabaw dla tensorflow. Ważne pojęcia (6)



Przeuczenie (overfiting): sytuacja, w której sieć dostosowuje się do nieistotnych fluktuacji danych uczących, co pogarsza generalizację.

Kodowanie wejścia

- Wejściem do sieci jest **wektor** (czyli ciąg liczb o ustalonej długości)
- W tym wektorze możemy zakodować wszystko:
  - obrazki (jak?)
  - teksty o ustalonej długości (jak?)
  - sytuację na planszy w Reversi (jak?)

Kodowanie **one-hot**

Sieci neuronowe lubią *rozwlekłe* kodowanie, w którym liczbę $i \in \{0, \ldots, N-1\}$ kodujemy jako $(0, 0, 0, \ldots, 1, \ldots, 0, 0)$ (jedynka na i-tej pozycji).

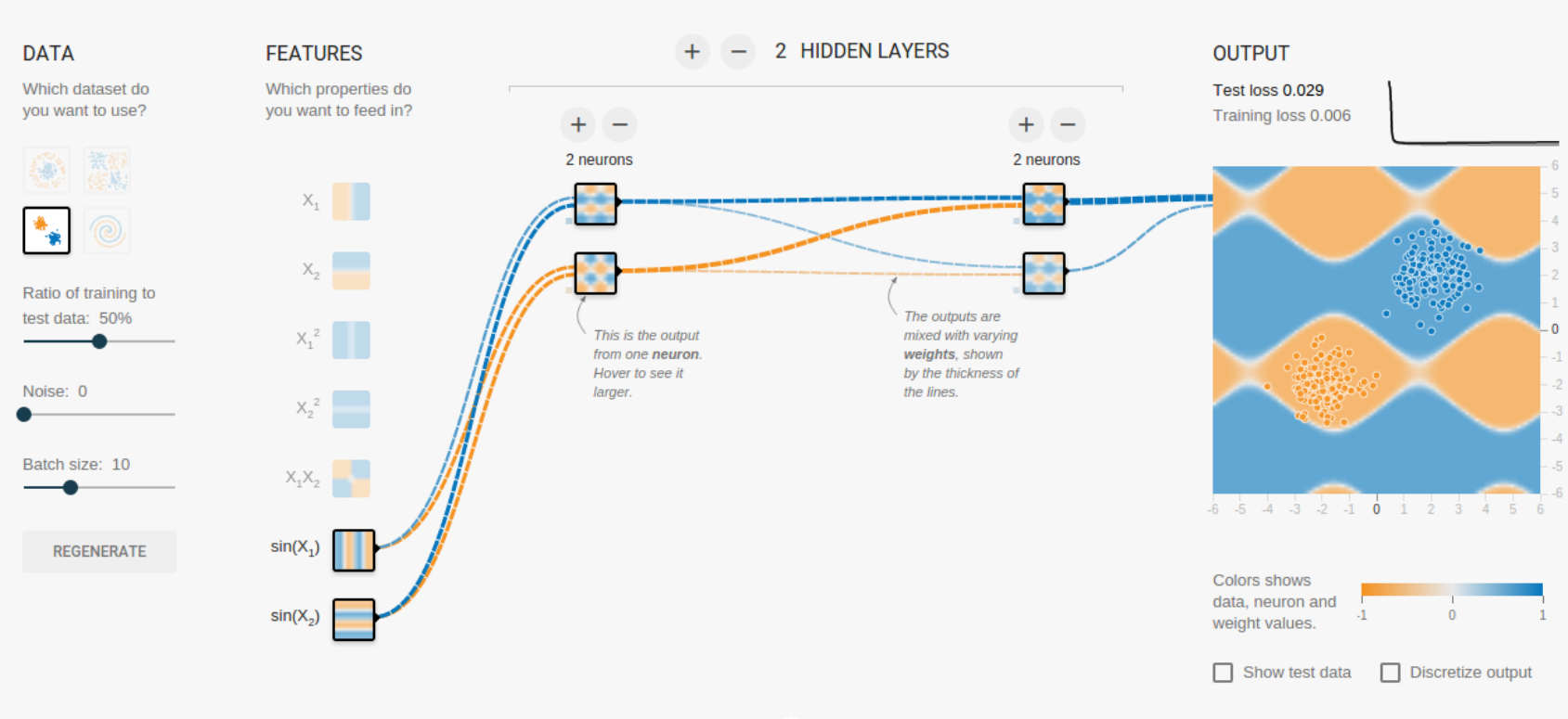
Kodowanie wejścia

- Zastanówmy się nad możliwymi kodowaniami obrazków, tekstów, fragmentów nagrań dźwiękowych, oraz planszy w reversi.
- Pamiętajmy, że możemy dowolnie tworzyć cechy dla przypadków testowych:
  - Kwantyzacja dla obrazów
  - Analiza Fouriera dla dźwięków
  - Tworzenie *pseudosłów* (rzeczownik, a-cja, ...)
  - ...

Uwaga

Dodając cechy możemy przyśpieszyć uczenie, ale możemy też *zasugerować* sieci naszą wizję świata. Np. cecha w Reversi: wynik jakiejś funkcji heurystycznej.

Sugerowanie cykliczności



Sieć w miarę poprawnie sklasyfikowała zbiór uczący, dobrze też go uogólnia, ale jest przekonana, że świat jest mozaiką. Nikt z nas, widząc te dane nie wyrobił sobie tego poglądu.

Softmax

- Często chcemy, żeby sieć decydowała o jednej z K opcji (zadanie klasyfikacji).
- Rozmywamy ten wybór, prosząc o podanie rozkładu prawdopodobieństwa dla wszystkich K opcji.
- To tzw. Softmax layer, która przypisuje prawdopodobieństwo zależne od wielkości pobudzenia.

Wzór:

$$\text{Sotfmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}}$$

Popatrzmy na to, jak działa funkcja Softmax.

Softmax

```
>>> softmax([1,2,3])
[0.09003, 0.24472, 0.66524]

>>> softmax([1,2,3,10]
[0.00012, 0.00033, 0.00091, 0.99863]

>>> softmax([3,4,4])
[0.15536, 0.42231, 0.42231]
```

Super łatwe sieci neuronowe

- Można wykorzystać bibliotekę **sklearn** (lub analogiczną), która implementuje **MLP** (czyli wielowarstwowy perceptron)
- Sieć definiujemy jednym konstruktorem z dużą liczbą parametrów (ale ufamy, że wartości domyślne są ok)

Super łatwe sieci neuronowe

Przygotowanie danych

```
from sklearn.neural_network import MLPClassifier
import random, pickle

# data: list of pairs (X,y)
# X: vector of floats/ints
# y in [v1,...,vk]

random.shuffle(data)
N = len(data) / 6
test_data = data[:N]
dev_data  = data[N:]

X = [x for (x,y) in dev_data]
y = [y for (x,y) in dev_data]
X_test = [x for (x,y) in test_data]
y_test = [y for (x,y) in test_data]
```

Super łatwe sieci neuronowe (2)

Uczenie sieci

```
# creating model
nn = MLPClassifier(hidden_layer_sizes=(60,60,10))

# training model
nn.fit(X,y)

print ('Dev score', nn.score(X,y))
print ('Test score', nn.score(X_test, y_test))

# writing model
with open('nn_weights.dat', 'w') as f:
    pickle.dump(nn, f)
```

Super łatwe sieci neuronowe (3)

Korzystanie z sieci

```
from sklearn.neural_network import MLPClassifier
import pickle

with open('nn_weights.dat') as f:
    nn = pickle.load(open(f))

x = data_vector

probabilities = nn.predict_proba([x])

prob0 = ys[0][0]
prob1 = ys[0][1]
```

Podsumowanie

Cons

- Oczywiście daje dużo mniejszą swobodę niż bardziej specjalizowane biblioteki.
- Nadaje się do tworzenia niezbyt dużych sieci
- Nie ma sieci splotowych, sieci rekurencyjnych, ...

Pros

- Bardzo prosta w użyciu i wystarczająco szybka
- Ten sam (prawie) interfejs dla różnych mechanizmów:
  - `from sklearn.neighbors import KNeighborsClassifier as Classifier`
  - `from sklearn.tree import DecisionTreeClassifier as Classifier`
  - `from sklearn.svm import SVC as Classifier`
  - ... (i jeszcze kilkanaście innych)