Sztuczna inteligencja. Trochę o sieciach neuronowych, Monte Carlo i bandytach, jak również o szukaniu szczęścia w niepewnym świecie

Paweł Rychlikowski

Instytut Informatyki UWr

14 maja 2018

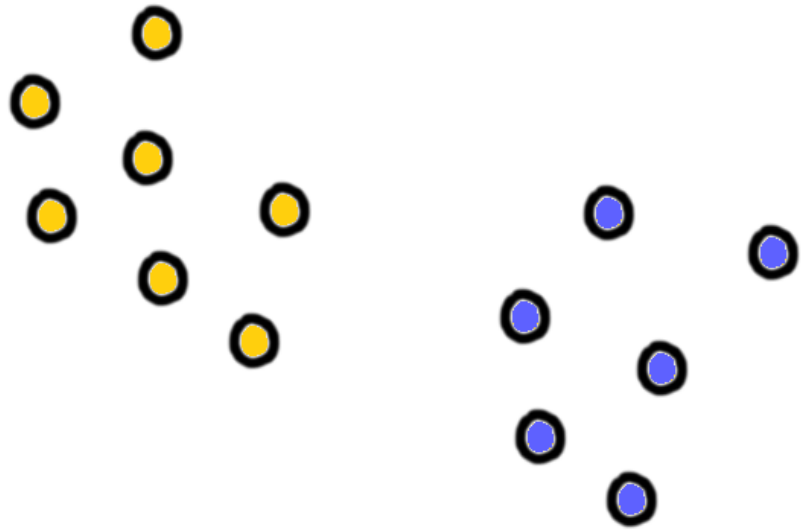
Sieci neuronowe w praktyce. Klasyfikacja

Ogólne założenia (przypadek dwóch klas)

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

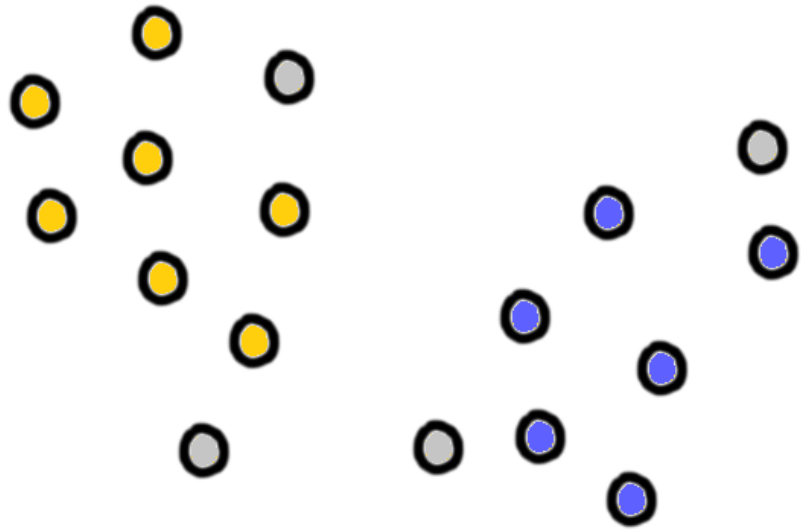
- Model neuronu: $f(w^T x + b)$ (czy wszyscy rozumiemy zapis?)
- To są klocki, z których składamy sieci.

Klasyfikacja w $\mathcal{R}^2$



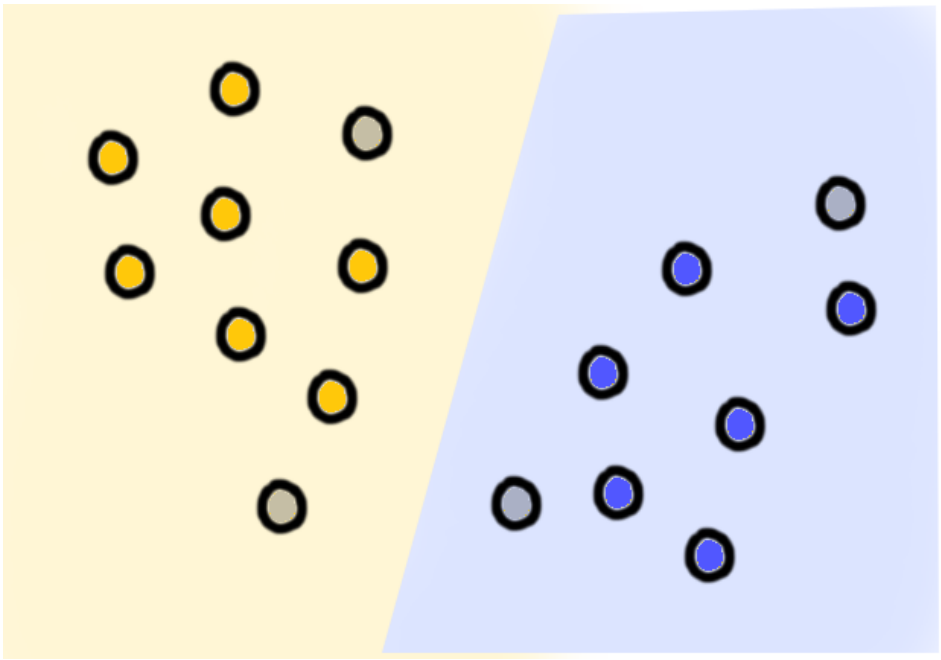
- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określać kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

Klasyfikacja w $\mathcal{R}^2$



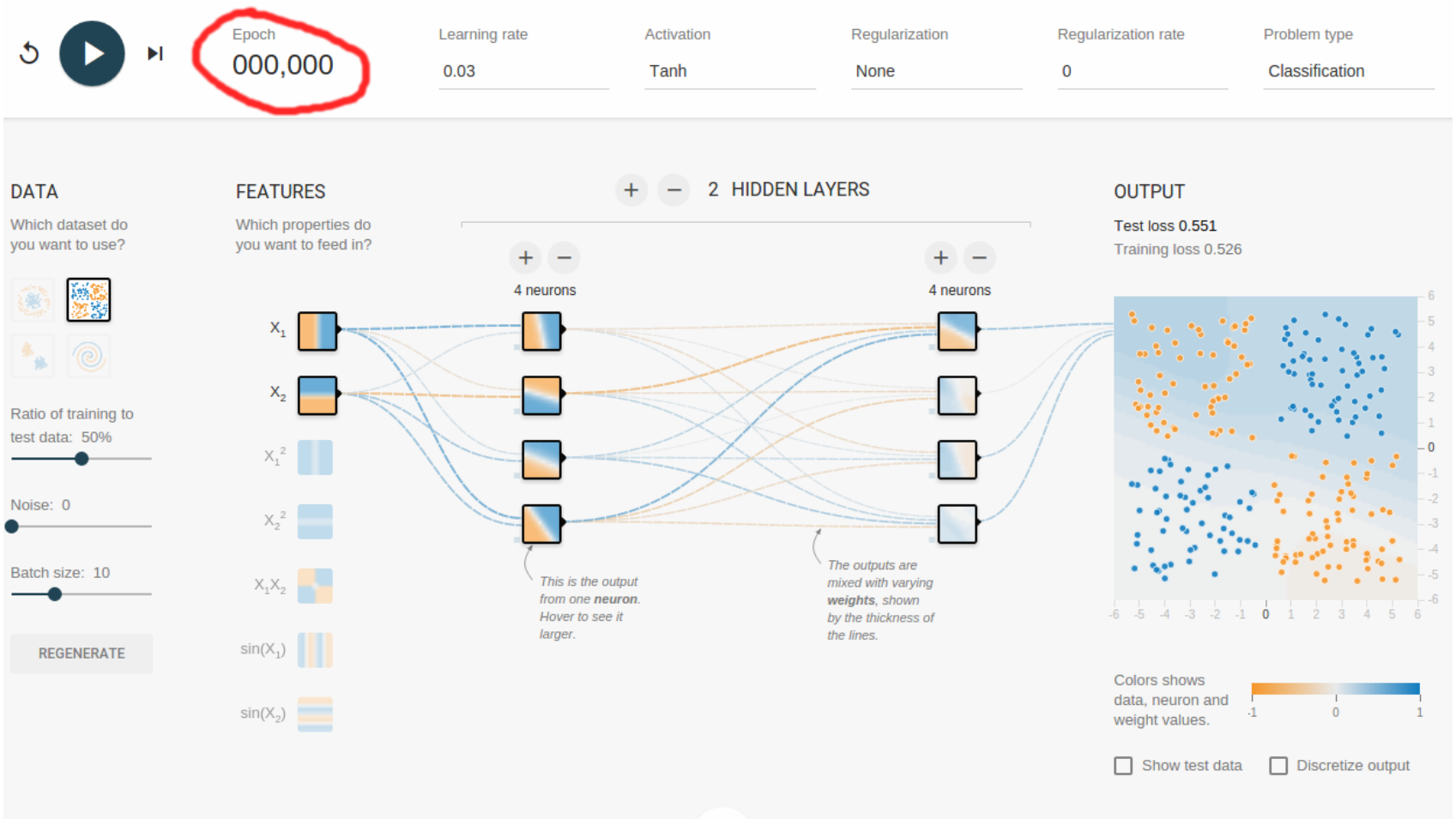
- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określać kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

Klasyfikacja w $\mathcal{R}^2$



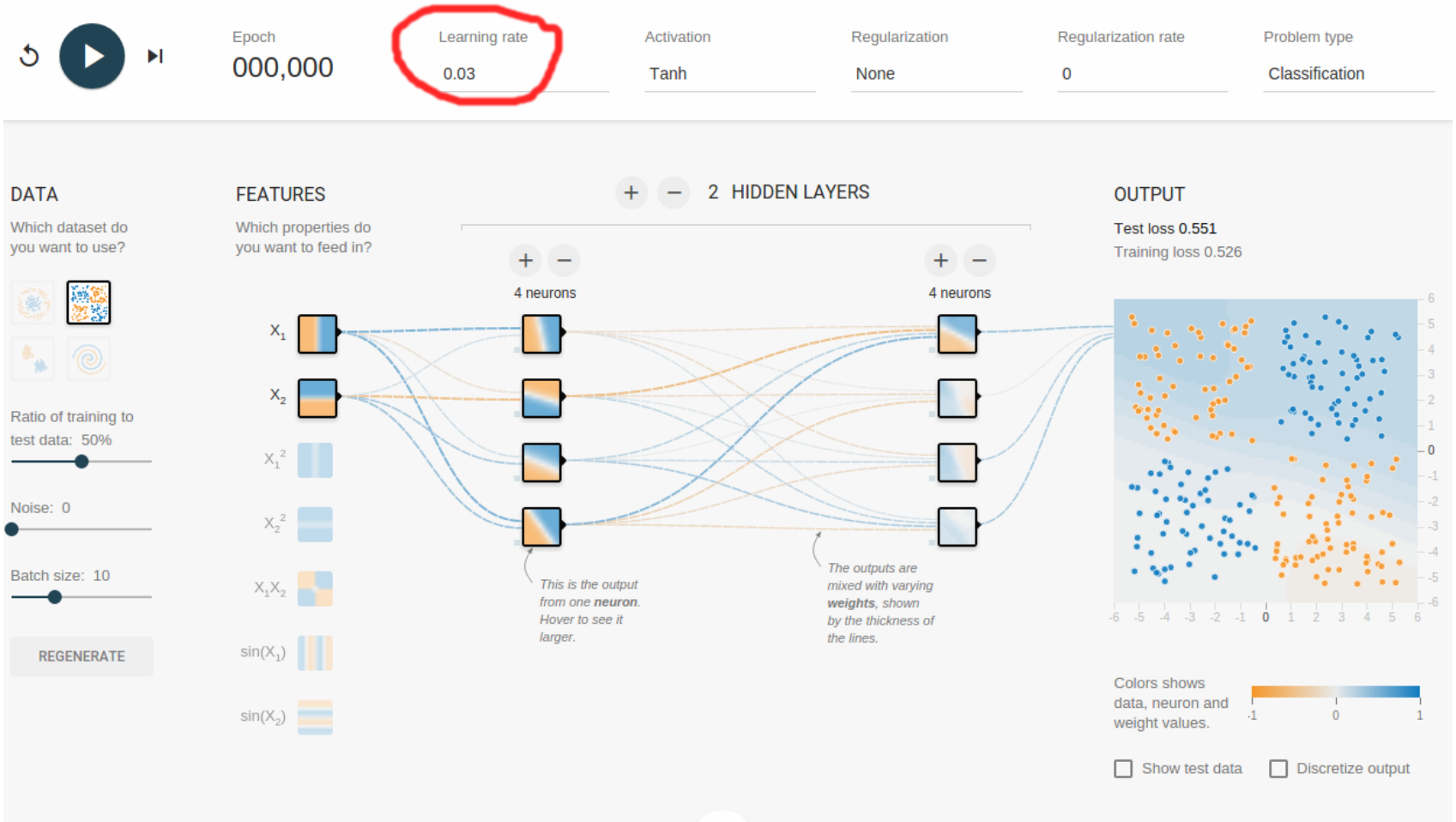
- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określać kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

Piaskownica dla tensorflow. Ważne pojęcia (1)



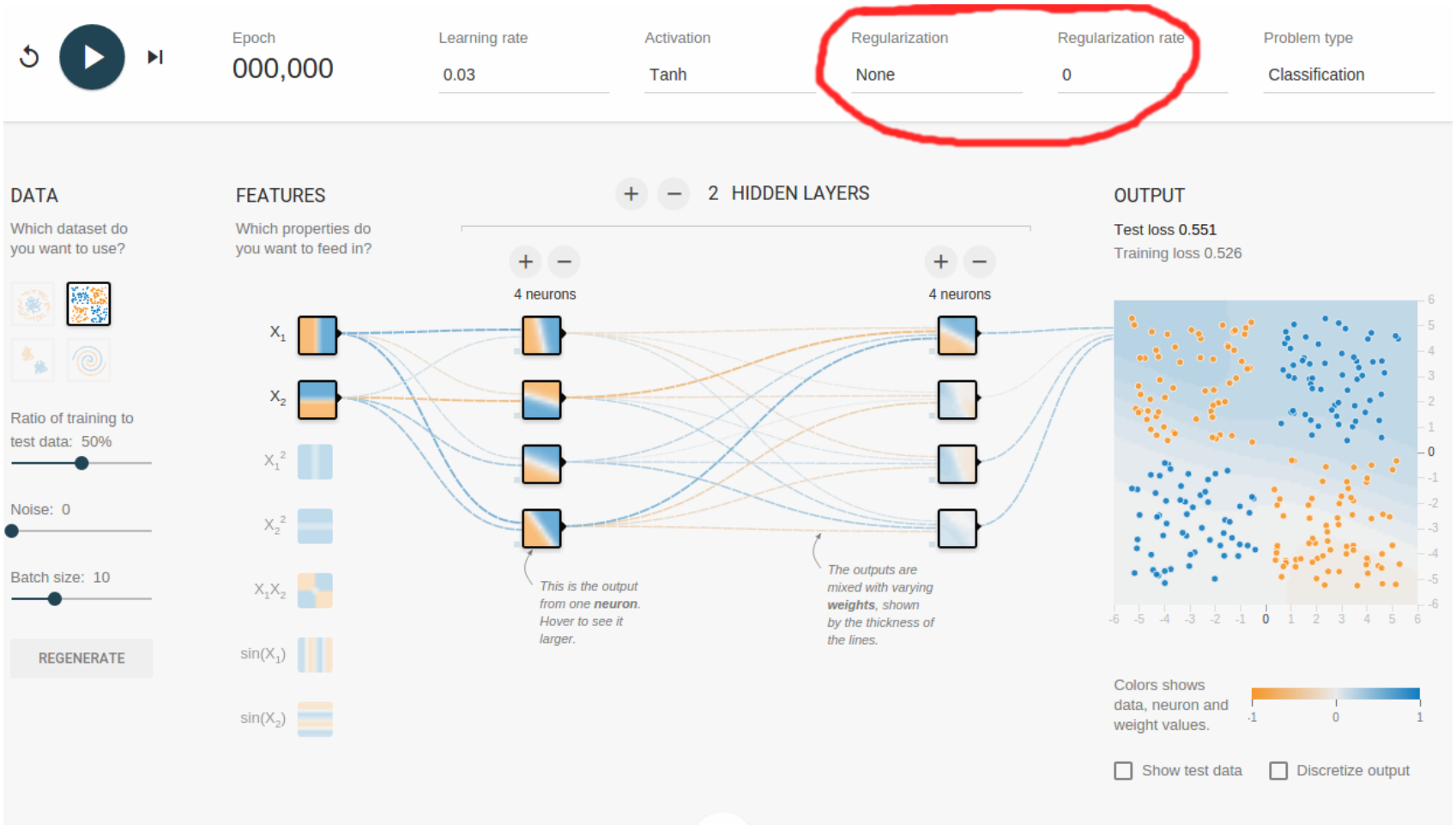
Epoka: etap uczenia, w którym uwzględnione są wszystkie dane uczące.

Piaskownica dla tensorflow. Ważne pojęcia (2)



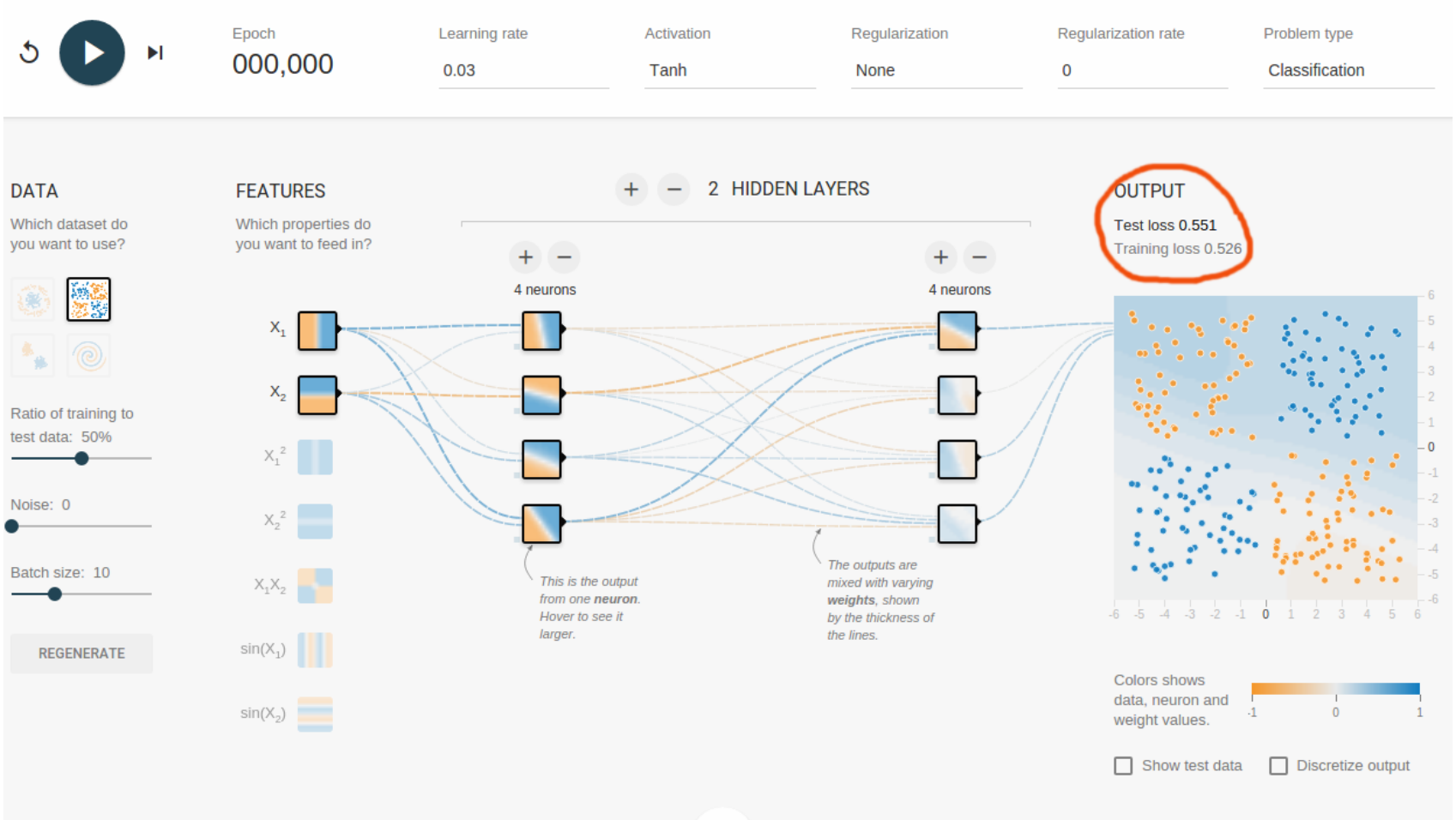
Learning rate: stała przez którą mnożone są delty wag. Za duża może dać chaotyczne zachowanie, za mała: bardzo wolny postęp.

Piaskownica dla tensorflow. Ważne pojęcia (3)



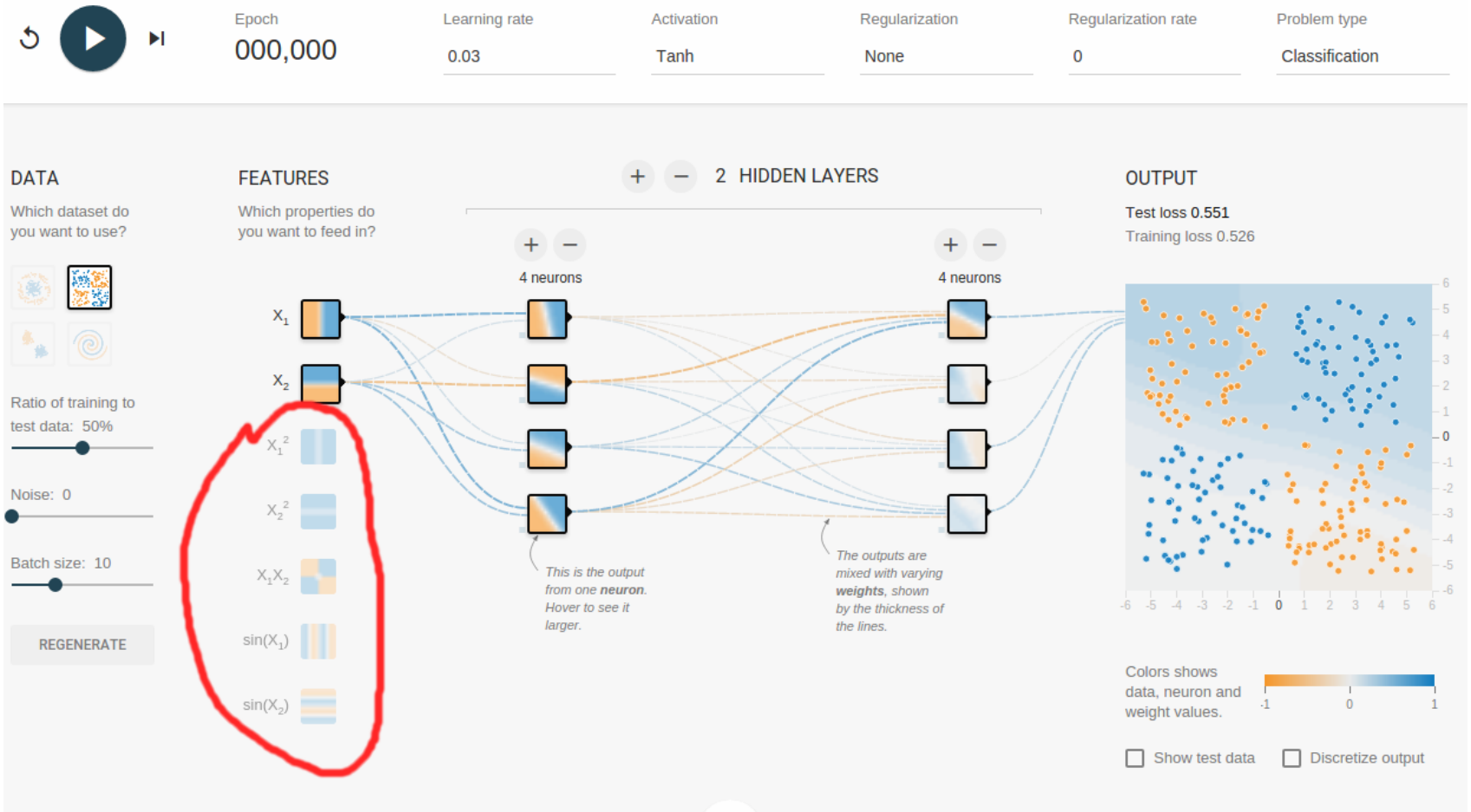
Regularyzacja: dołożenie do uczenia wymagania, by wagi nie były zbyt duże. Może dać większą stabilność uczenia (zob. tablica).

Piaskownica dla tensorflow. Ważne pojęcia (4)



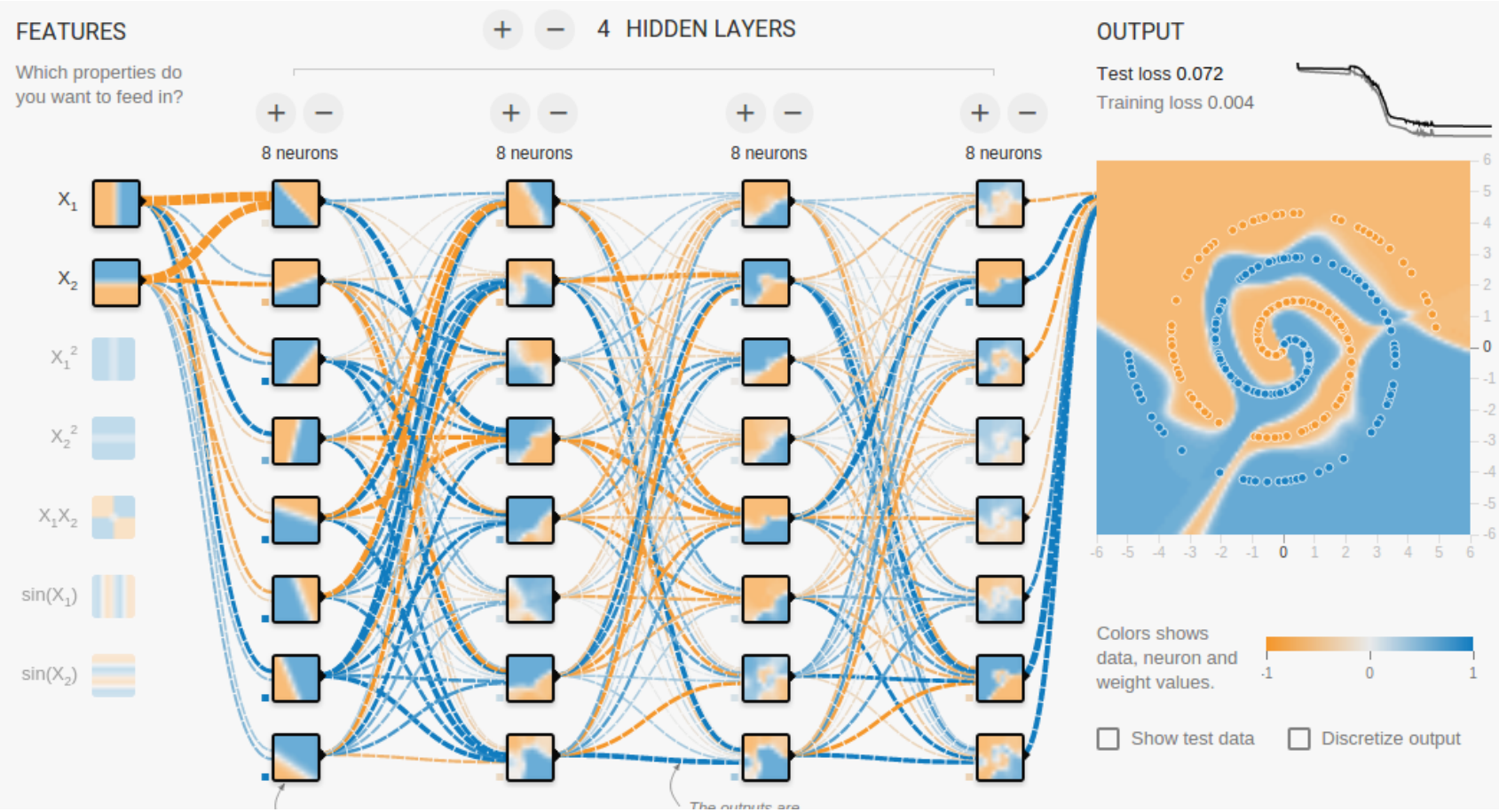
Test loss/training loss: wartość „straty” dla zbioru testowego i uczącego (oczywiście pierwsza zawsze większa).

Piaskownica dla tensorflow. Ważne pojęcia (5)



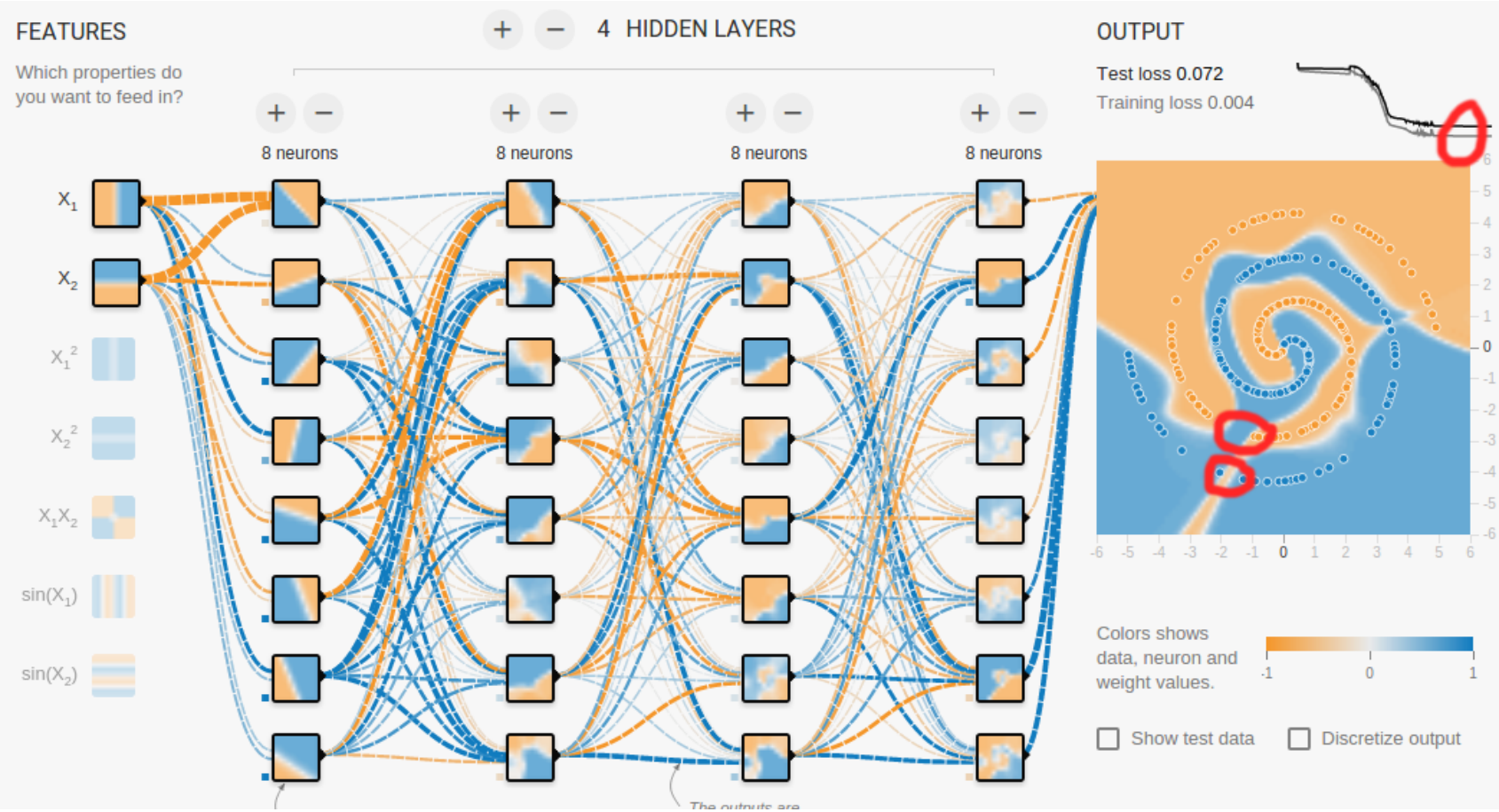
Feature engineering: proces tworzenia własnych cech dla konkretnych przypadków. Dobre cechy mają związek z zadaniem.

Piaskownica dla tensorflow. Ważne pojęcia (6)



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

Piaskownica 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) (zapominamy na razie o sieciach konwolucyjnych)
- 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).

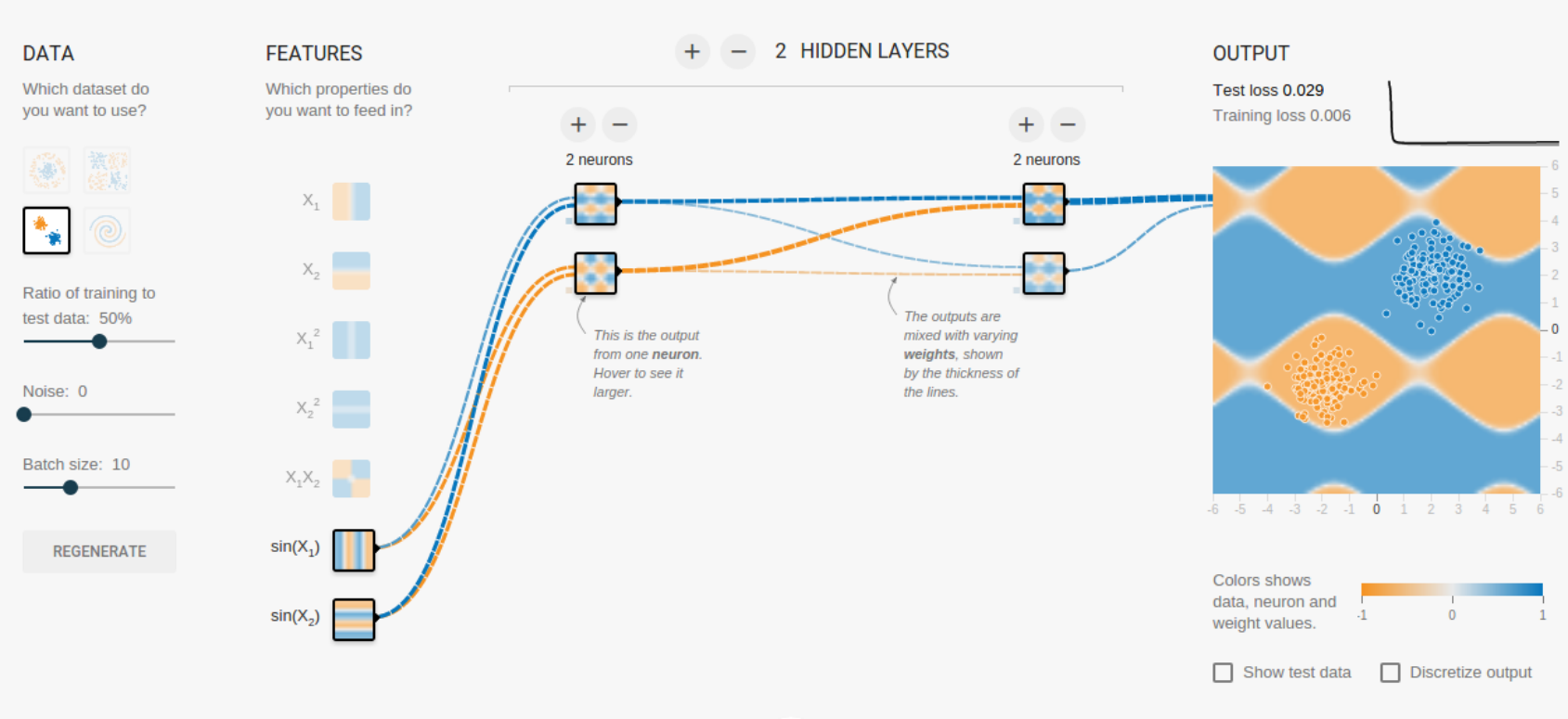
Kodowanie wejścia

- 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.

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 dev_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 konwolucji, 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)

Eksploracja i eksploatacja

- Standardowy dylemat agenta działającego w nieznanym środowisku:
  1. Maksymalizować swoją korzyść biorąc pod uwagę aktualną wiedzę o świecie.
  2. Starać się dowiedzieć więcej o świecie, być może ryzykując nieoptymalne ruchy.
- Pierwsza strategie to eksploatacja, druga to eksploracja.

Jednoręki bandyta

Źródło: Wikipedia

Po pociągnięciu za rączkę, pojawia się wzorek, który (potencjalnie) oznacza naszą niezerową wypłatę.

Wieloręki bandyta

- Mamy wiele tego typu maszyn.
- Możemy zapomnieć o wzorkach, maszyny po prostu generują wypłatę, zgodnie z nieznanym rozkładem.
- Bardzo wyraźnie widać dylemat eksploracja vs eksploatacja.

Wieloręki bandyta. Przykładowe strategie

- **Zachłanna**: każda rączka po razie, a następnie... ta która dała najlepszy wynik.
- **ε-zachłanna**: rzucamy monetą. Z $p = \varepsilon$ wykonujemy ruch losową rączką, z $p = 1 - \varepsilon$ – wykonujemuy ruch rączką, która ma naljepszy średni wynik do tej pory.
- **Optymistyczna wartość początkowa**: inny sposób na zapewnienie eksploracji. Na początku każdy wybór obniża atrakcyjność danego bandyty.

Upper Confidence Bound

- Wybieramy akcję a (bandytę) maksymalizującą:

$$Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}$$

gdzie: Q_t to wartość akcji w czasie t, N_t – ile razy dana akcje była wybierana (do momentu t)

- Zwróćmy uwagę, że jak akcja nie jest wybierana, to prawy składnik powoli rośnie. Akcja wybierana natomiast traci „premię eksploracyjną", na początku w szybkim tempie (wzrost mianownika).

Uwaga

Bardzo powszechnie używana strategia! (np. w AlphaGo)

Monte Carlo Tree Search

Algorytm odpowiedzialny za przełom w:

a. W grze w Go

b. W General Game Playing

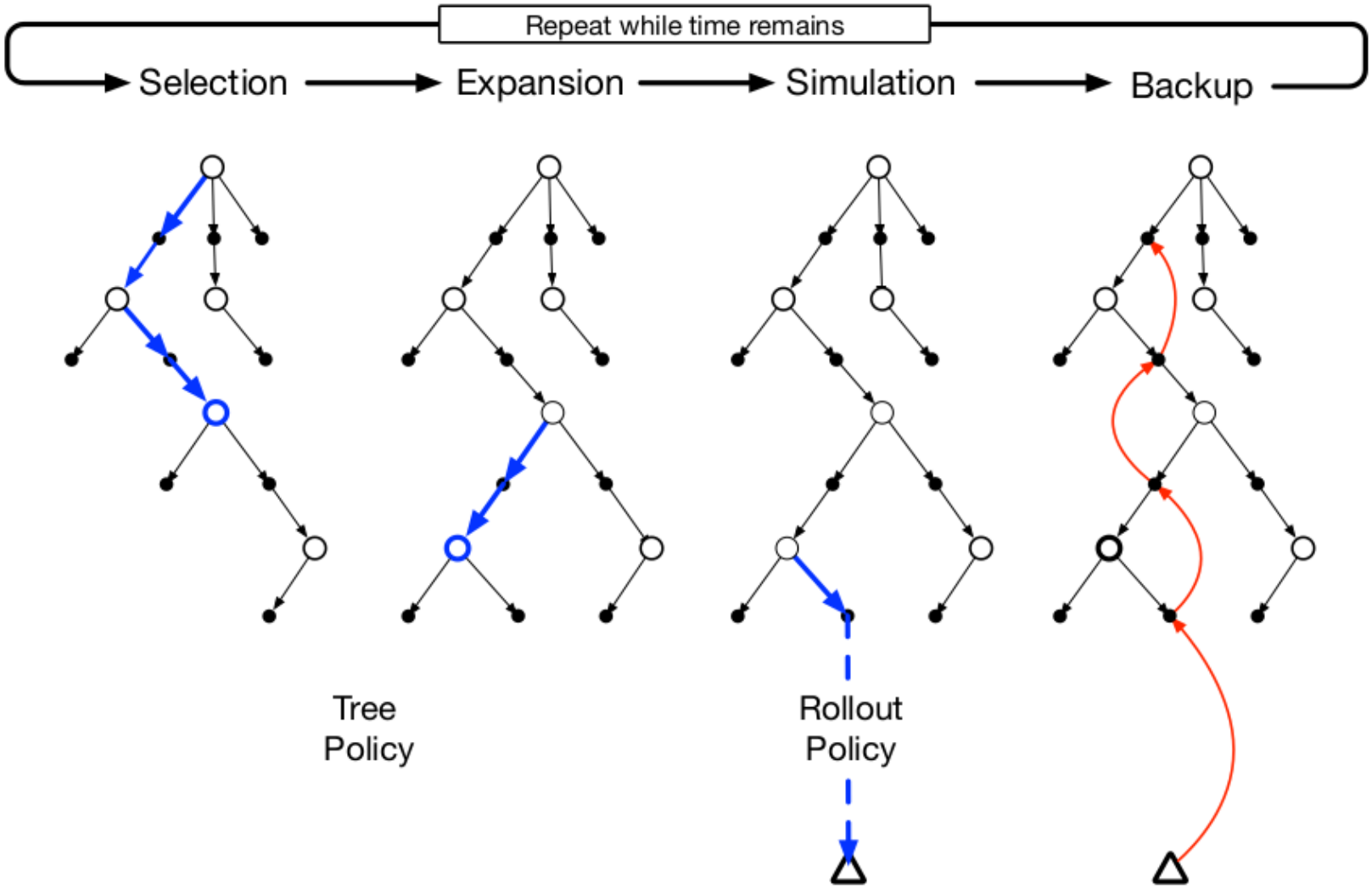
Główne idee

1. Oceniamy sytuację wykonując symulowane rozgrywki.
2. Budujemy drzewo gry (na początku składające się z jednego węzła – stanu przed ruchem komputera)
3. Dla każdego rozwiniętego węzła utrzymujemy statystyki, mówiące o tym, kto częściej wygrywał gry rozpoczynające się w tym węźle
4. Rozwijamy wybrany węzeł (UCB) dodając jego dzieci i przeprowadzając rozgrywkę.

MCTS. Podstawowe operacje

1. **Selection**: wybór węzła do rozwinięcia
2. **Expansion**: rozwinięcie węzła (dodanie kolejnych stanów)
3. **Simulation**: symulowana rozgrywka (zgodnie z jakąś polityką), zaczynające się od wybranego węzła
4. **Backup**: uaktualnienie statystyk dla rozwiniętego węzła i jego przodków

MCTS. Rysunek



MCTS. Dodatkowe uwagi

- Rozgrywka nie musi być prostym losowaniem, p-stwo ruchu może zależeć od jego (szybkiej!) oceny.
- Im więcej symulacji, tym lepsza gra – precyzyjne sterowanie trudnością i czasem działania.