Mini-batch SGD

Uczenie odbywa się w następującej pętli treningowej:

1. Wybierz partię próbek x i odpowiednich celów y.
2. Podaj na wejście sieci x (krok nazywany *forward pass*)), aby uzyskać prognozy y_{pred}.
3. Oblicz stratę sieci czyli błąd między y_{pred} i y.
4. Oblicz gradient funkcji błędu f(W) i zmodyfikuj wszystkie wagi: $W_1 = W_0 - \alpha \cdot \nabla f(W_0)$
5. Jeżeli to konieczne (błąd jest nadal duży) – wróć do punktu 1.

Optymalizacja gradientowa

Uczenie maszynowe polega na aktualizacji współczynników metodą gradientową. Dostępnych jest kilka wariantów tej metody w zależności od wielkości próbki treningowej.

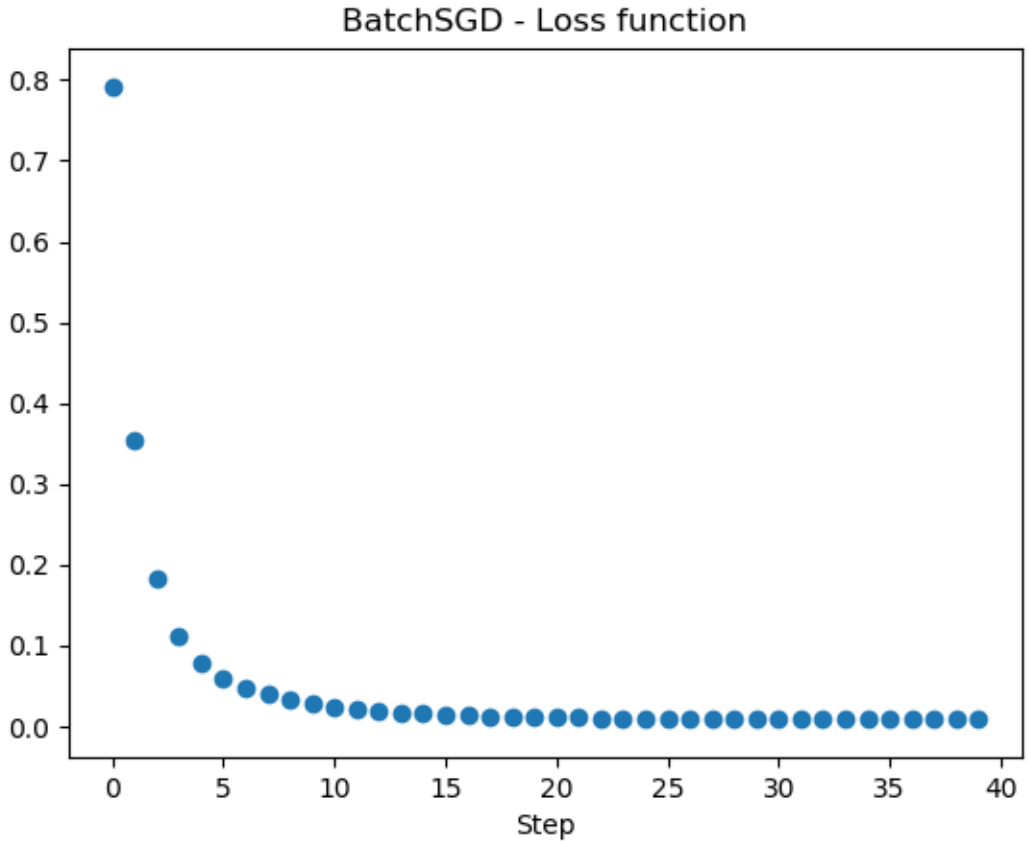
Batch Gradient Descent – gradient funkcji straty obliczany jest dla całego zestawu treningowego w każdej epoce.

Minusy takiego rozwiązania:

- Wymaga załadowania całego zestawu danych do pamięci.
- Możliwe utknięcie w minimach lokalnych – mniejsza szansa na znalezienie minimum globalnego.

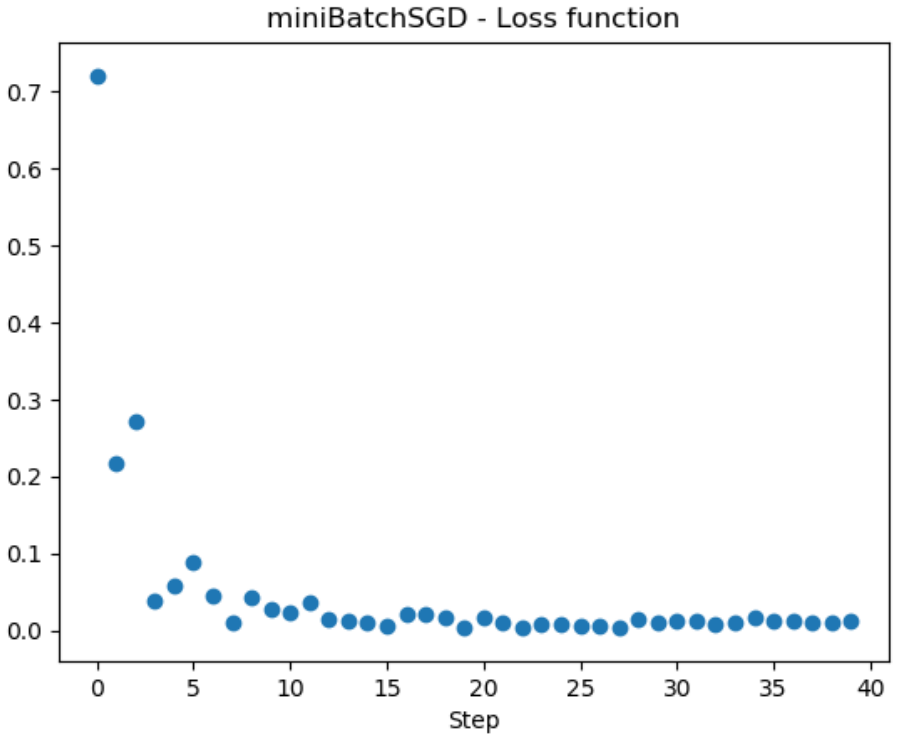
Optymalizacja gradientowa

Batch Gradient Descent – zmiana błędu:



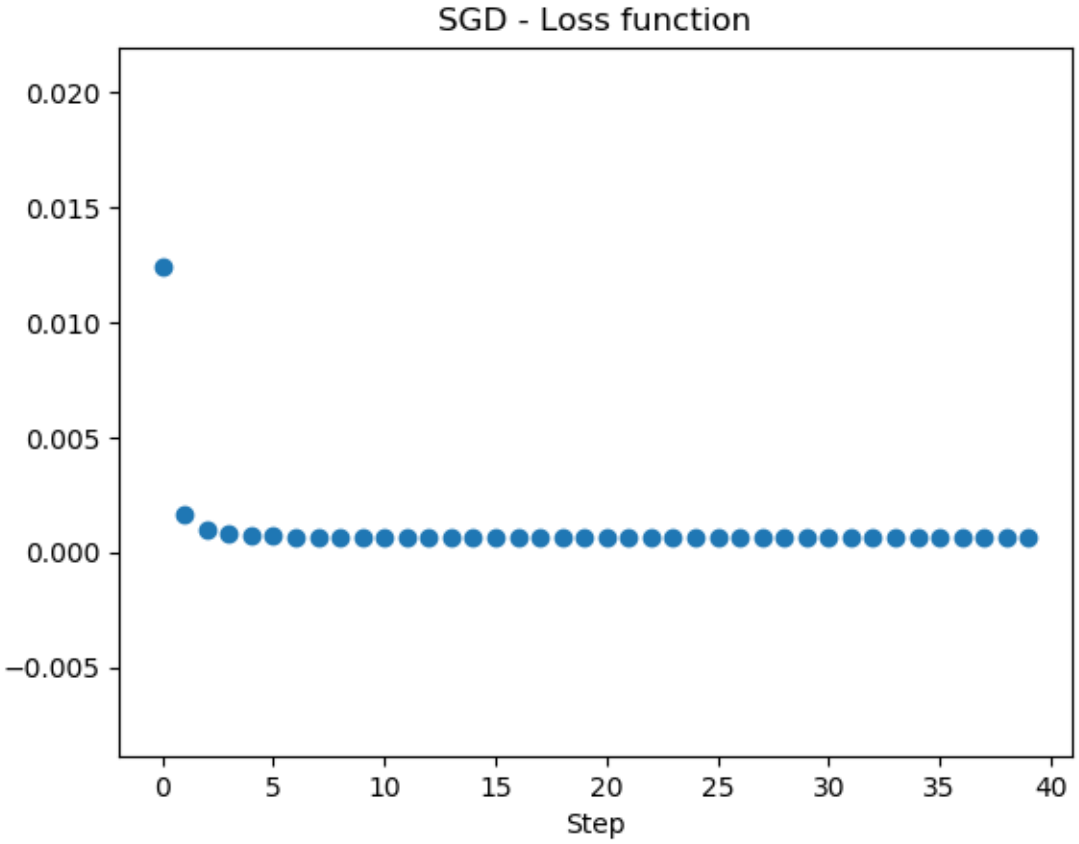
Optymalizacja gradientowa

Mini-batch gradient descent – w tym przypadku parametry są aktualizowane dla pewnej partii danych treningowych (tzw. batch).



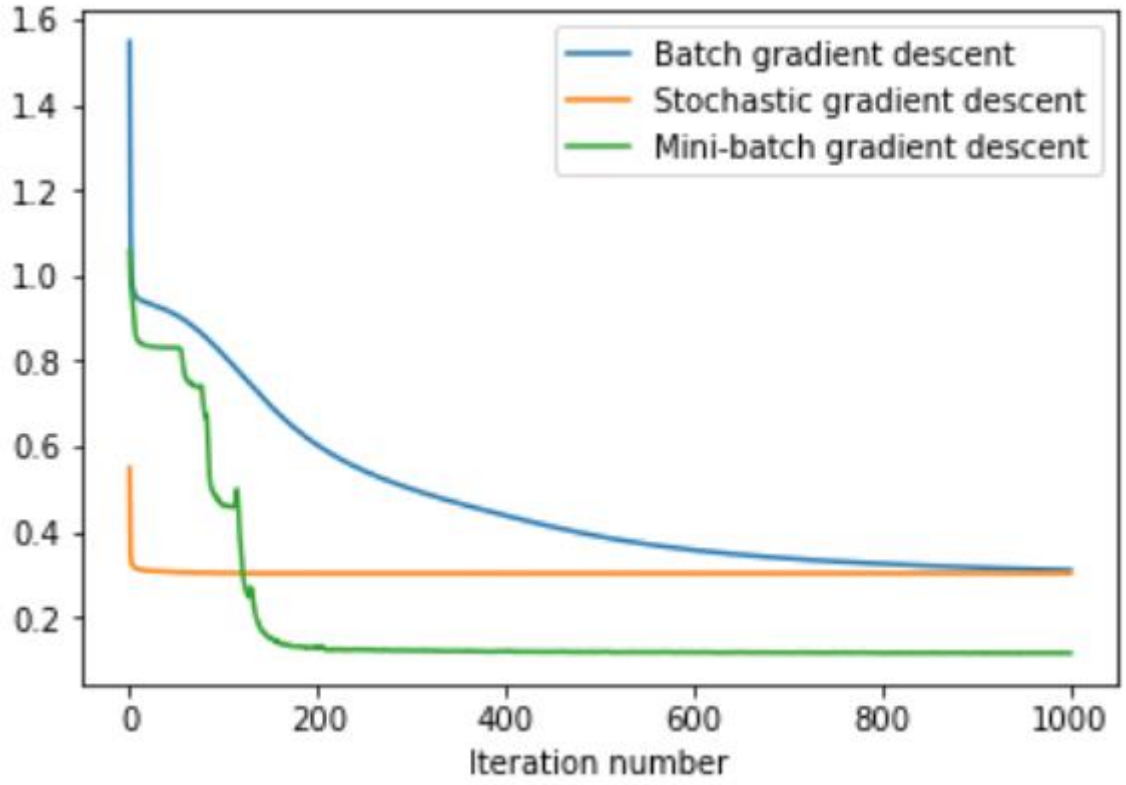
Optymalizacja gradientowa

True Stochastic Gradient Descent – gradient funkcji straty obliczany jest dla pojedynczego elementu z zestawu treningowego w każdej epoce.



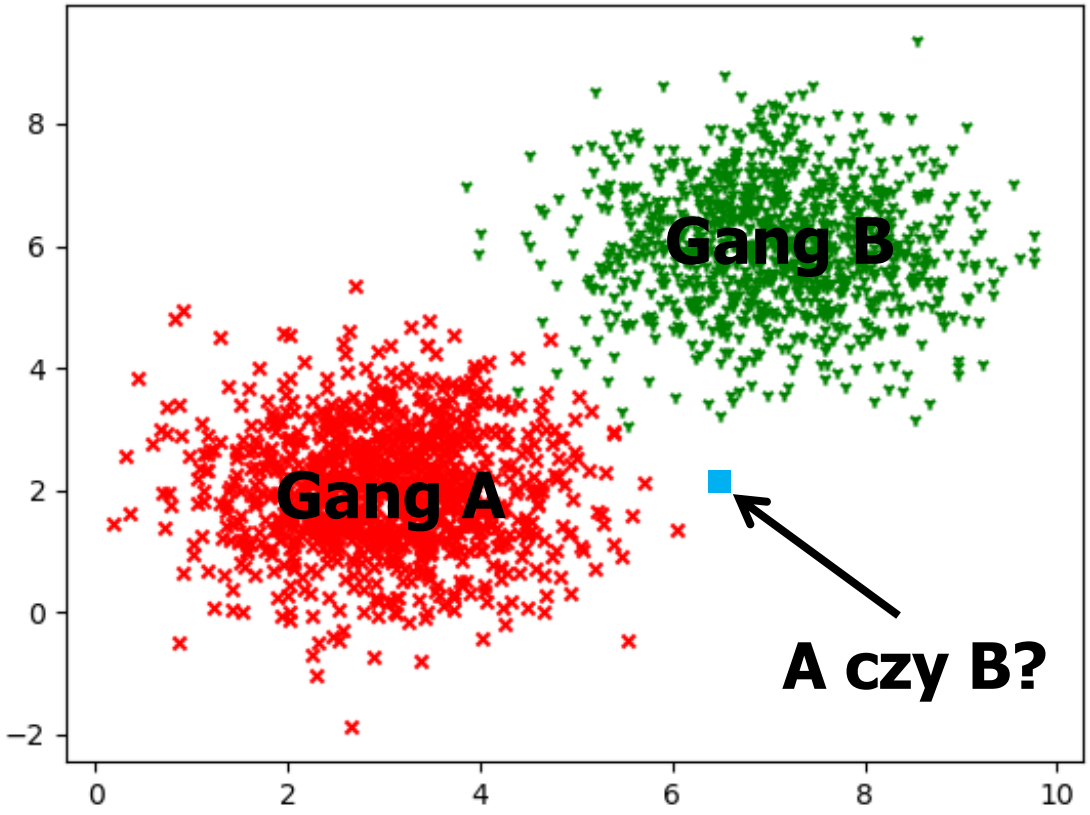
Optymalizacja gradientowa

Podsumowanie:



Regresja logistyczna

Rozważmy teraz zbiór danych:

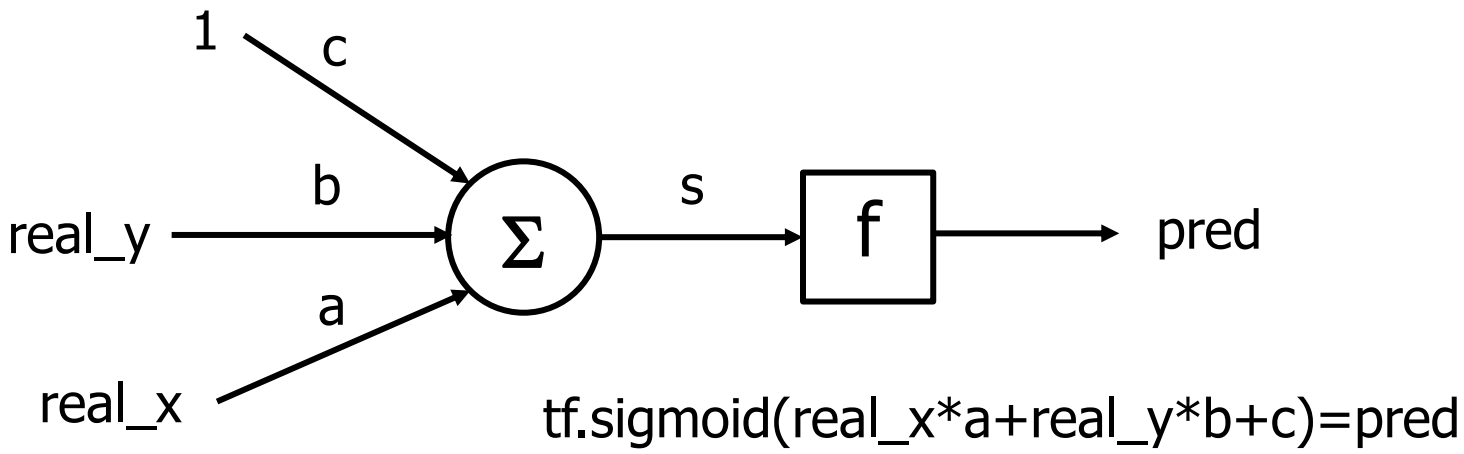


Dwa gangi – regresja logistyczna

Rozwiązujemy problem za pomocą jednego neuronu o jednym wejściu.

```
pred=tf.sigmoid(a*real_x+b*real_y+c)
```

Wagami neuronu są parametry a, b i c.



Regresja logistyczna

Definiujemy model:

```
model = Sequential()

model.add(Dense(units = 1, use_bias=True,
input_dim=2, activation = "sigmoid"))

opt = keras.optimizers.Adam(learning_rate=0.1)
#opt = keras.optimizers.SGD(learning_rate=0.001)

model.compile(loss='binary_crossentropy',optimizer=
opt)

model.summary()
```

Regresja logistyczna

Pętla ucząca:

```
epochs = 1000
h = model.fit(data_points,labels, verbose=1,
epochs=epochs, batch_size=1000)
```

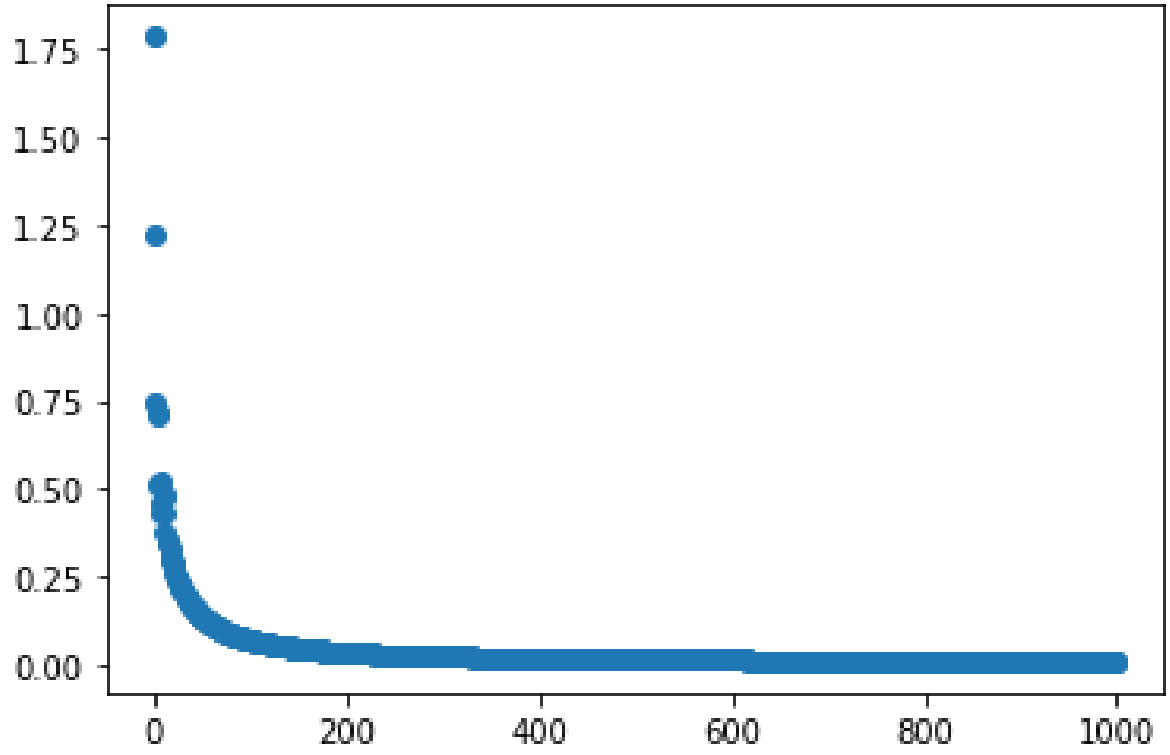
Błąd i wykres jego zmian:

```
Loss = h.history['loss’]

plt.scatter(np.arange(epochs),Loss)
plt.show()
```

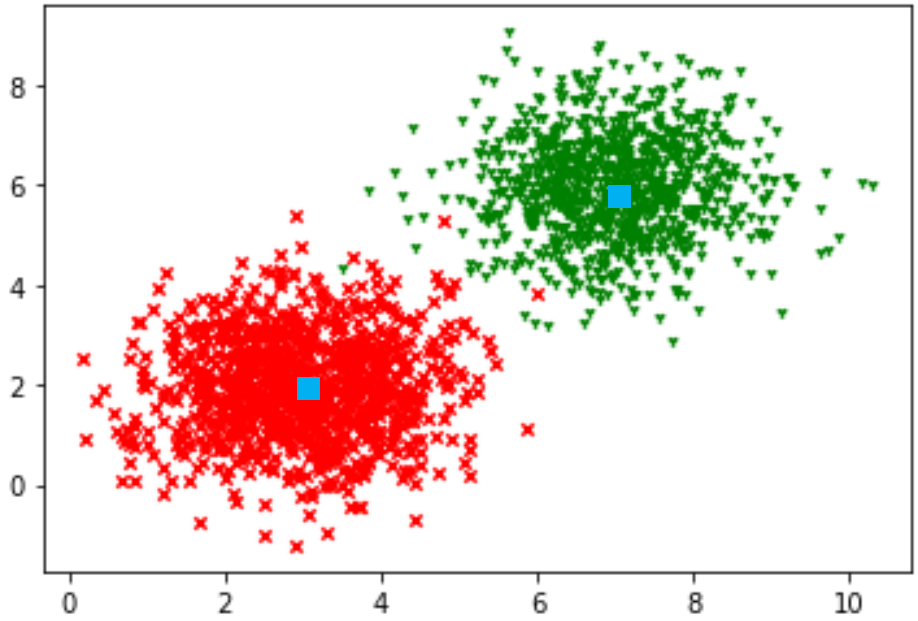
Regresja logistyczna

Zmiany błędu:



Regresja logistyczna

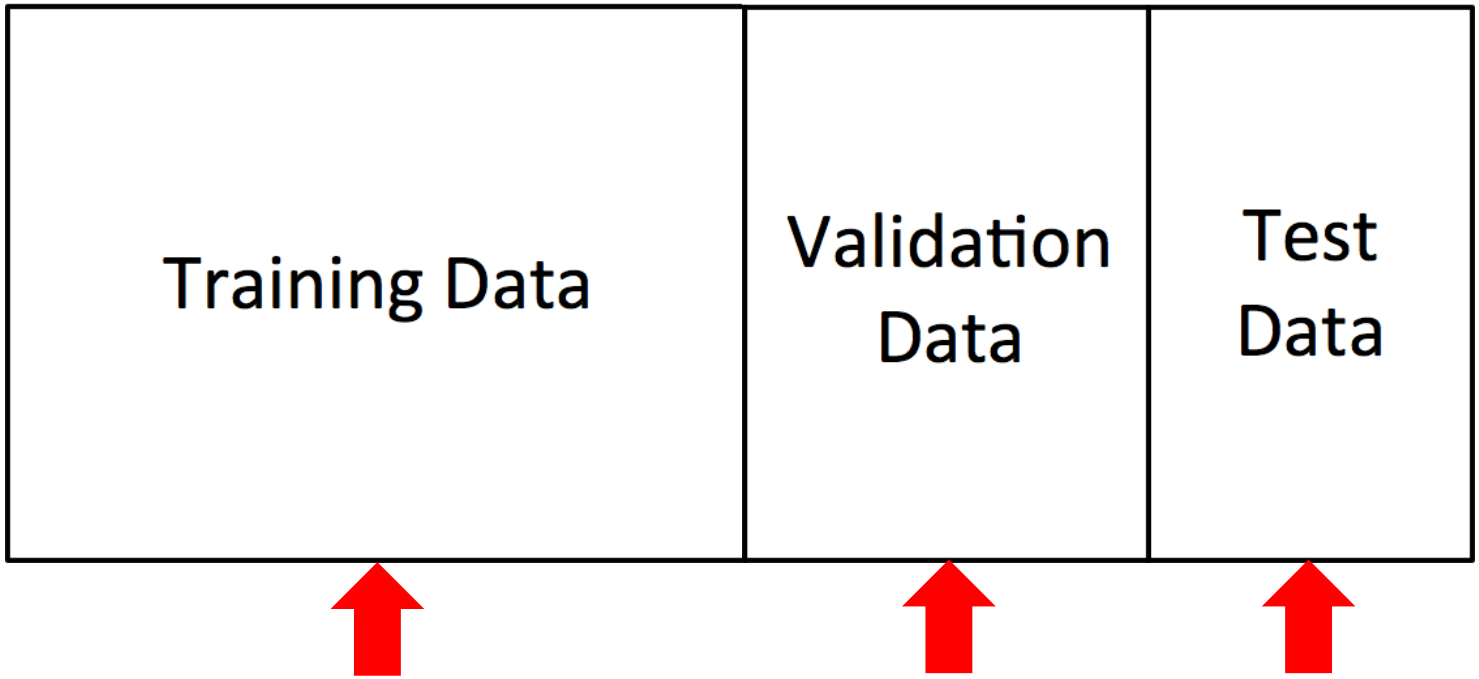
Test:




```
model.predict([[7.0,6.0]])
→ array([[0.9997442]], dtype=float32)
```

```
model.predict([[3.0,2.0]])
→ array([[0.0002591]], dtype=float32)
```

Dane


Training Data
Validation Data
Test Data
uczenie modelu
walidacja modelu
testowanie modelu

Dane

Dane do walidacji?

Opracowanie modelu zawsze wymaga dostrojenia jego konfiguracji: na przykład, musimy dobrać liczbę warstw sieci lub rozmiar czyli ilość neuronów (liczby te nazywanych hiperparametrami modelu, parametry modelu to jego wagi).

Dostrajanie modelu przeprowadzamy wykorzystując jako sygnał sprzężenia zwrotnego działanie modelu na danych walidacyjnych.

Dostrajanie modelu jest formą uczenia się: wyszukiwanie dobrej konfiguracji w pewnej przestrzeniach hiperparametrów.

Dane

W rezultacie, długotrwałe dostrajanie modelu w oparciu o jego działanie na zbiorze walidacyjnym może skutkować przeuczeniem (overfitting) na tym zbiorze, nawet jeżeli nasz model nigdy nie był bezpośrednio na nim uczony.

Za każdym razem, gdy się dostroimy jakiś hiperparametr modelu w oparciu o wydajność modelu na danych walidacyjnych, pewna informacja o tych danych przenika do modelu (information leaks).

Jeżeli takie dostrajanie powtórzmy wielokrotnie to dużo informacji o danych walidacyjnych wycieknie do modelu. Stąd możliwość przeuczenia.

Dane

Dane do testów?

Ponieważ dostrojenie może prowadzić do przeuczenia na danych walidacyjnych bardzo ważne są dane testowe z którymi model nie miał nigdy kontaktu (nawet niebezpośrednio).

Dane testowe służą do sprawdzenia tego jak model uogólnia czyli jak zachowuje się na danych, których nigdy "nie widział".

Regresja logistyczna

Podział na zbiór treningowy i walidacyjny:

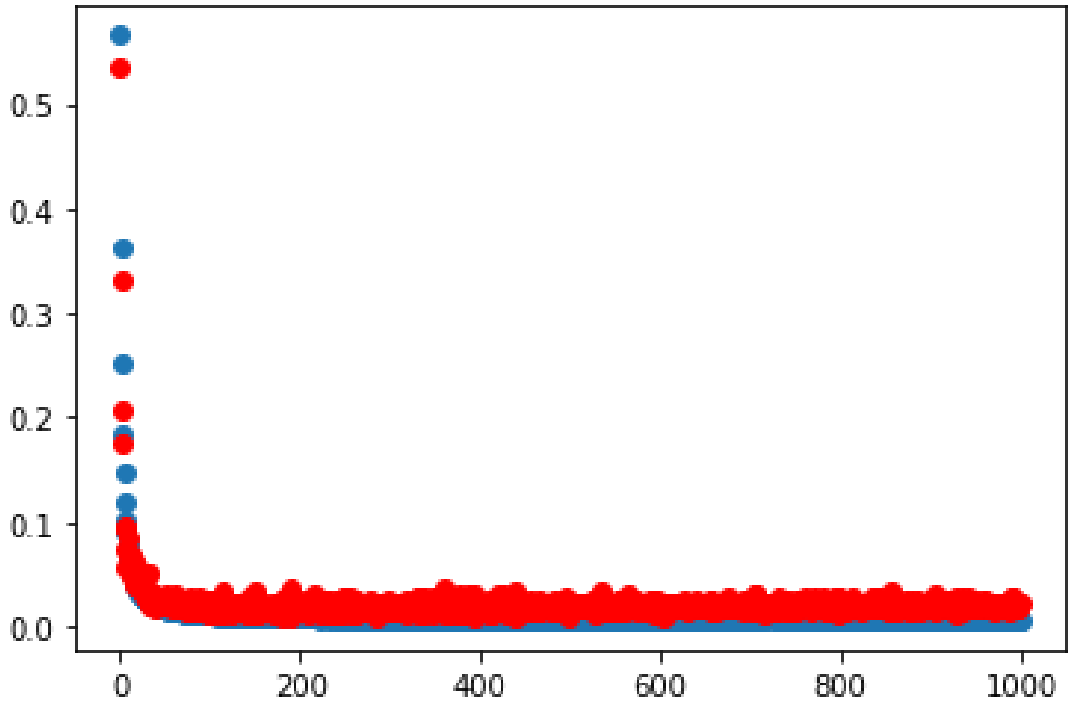
```
h = model.fit(data_points,
              labels,
              verbose=0,
              validation_split=0.2,
              epochs=epochs,
              batch_size=100)
```

Czyli 20% danych to dane walidacyjne.

Regresja logistyczna

Wykresy:

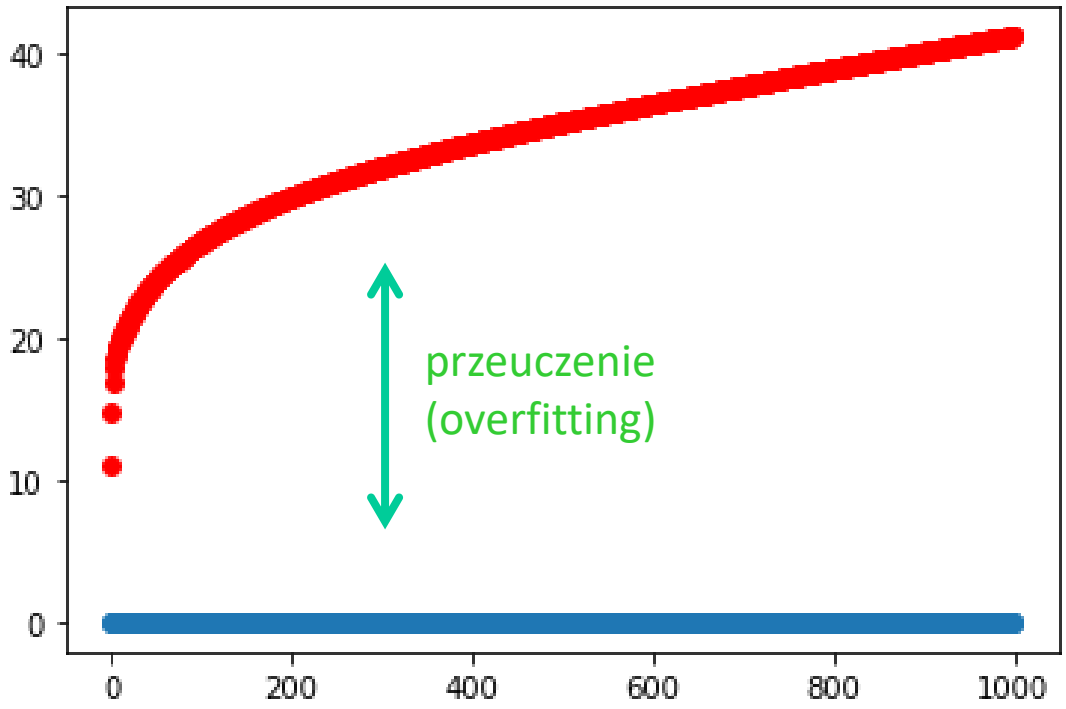
```
plt.scatter(np.arange(epochs),h.history['loss'])
plt.scatter(np.arange(epochs),h.history['val_loss']
,c='r')
plt.show()
```



Regresja logistyczna

Zbyt mały zbiór treningowy:

```
h = model.fit(data_points,labels,verbose=0,
              validation_split=0.6,
              epochs=epochs,
              batch_size=100)
```



Optymalizacja gradientowa

Istnieje wiele wariantów SGD, które różnią się np. tym, że podczas obliczania następnej aktualizacji parametrów uwzględniają także poprzednie wartości gradientów, a nie tylko ich bieżące wartości.

Jest to np. SGD z członem momentum, a także Adagrad, RMSProp i kilka innych.

Takie warianty nazywane są metodami optymalizacji lub **optymalizatorami**.