Metody Inżynierii Wiedzy

Systemy uczące się - podejście klasyczne

Dr inż. Michał Majewski

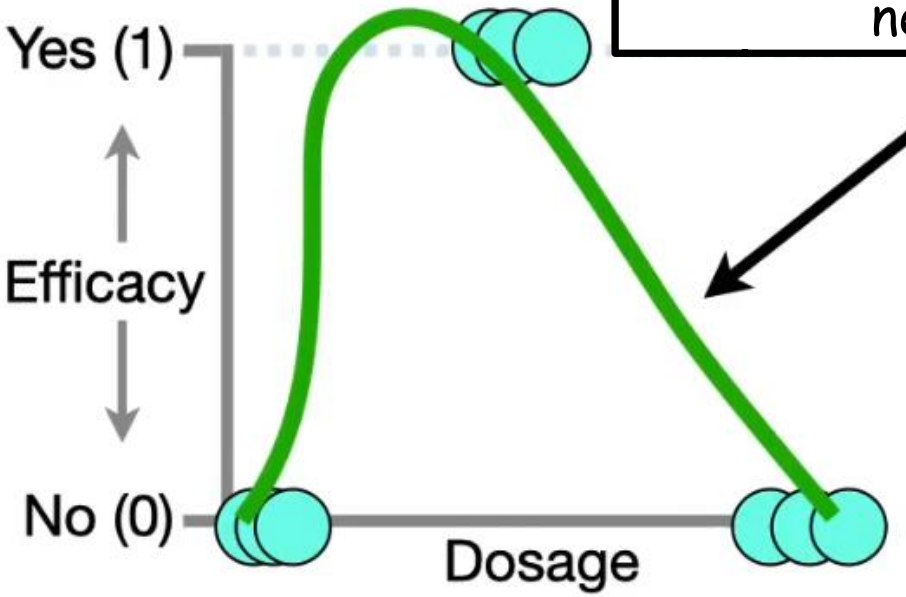
mmajew@pjwstk.edu.pl

materiały: *ftp*(*public*) : *//mmajew/MIW*

Podstawowe idee sieci neuronowych

Dawka leku
vs.
Efektywność leczenia wirusa

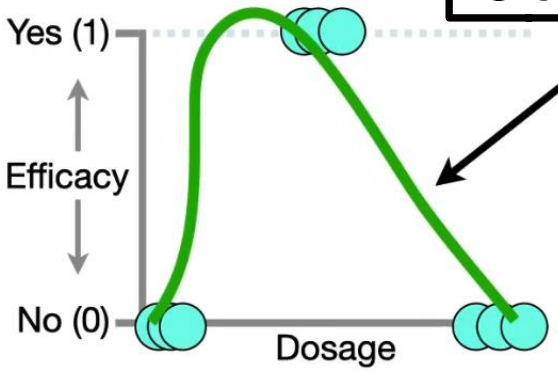
Przewidywania sieci neuronowej



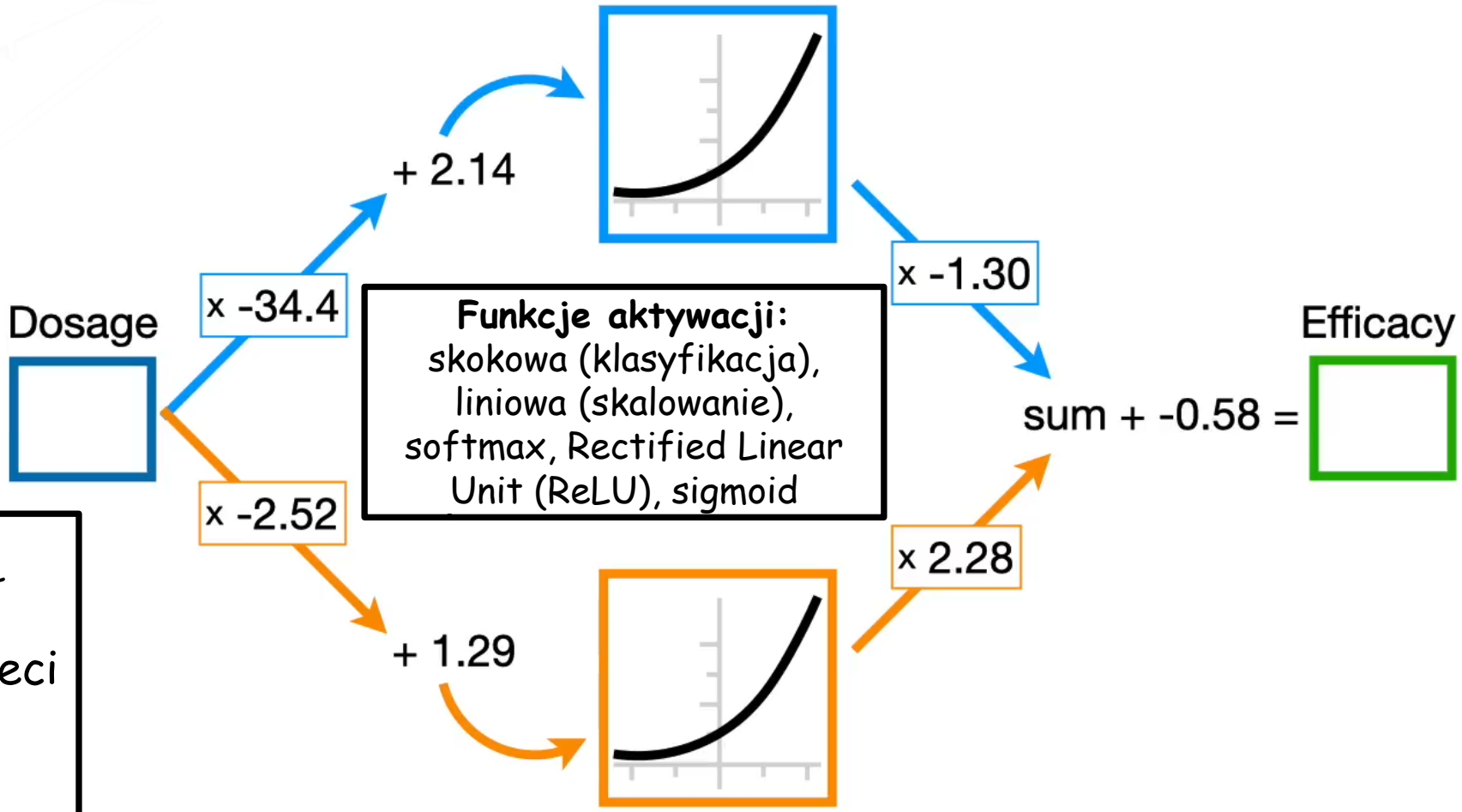
https://www.youtube.com/watch?v=CqOfi41LfDw&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=2 StatQuest with Josh Starmer

Podstawowe idee sieci neuronowych

Przewidywania sieci neuronowej



Sieć neuronowa : 3 warstwy, węzły, połączenia między węzłami



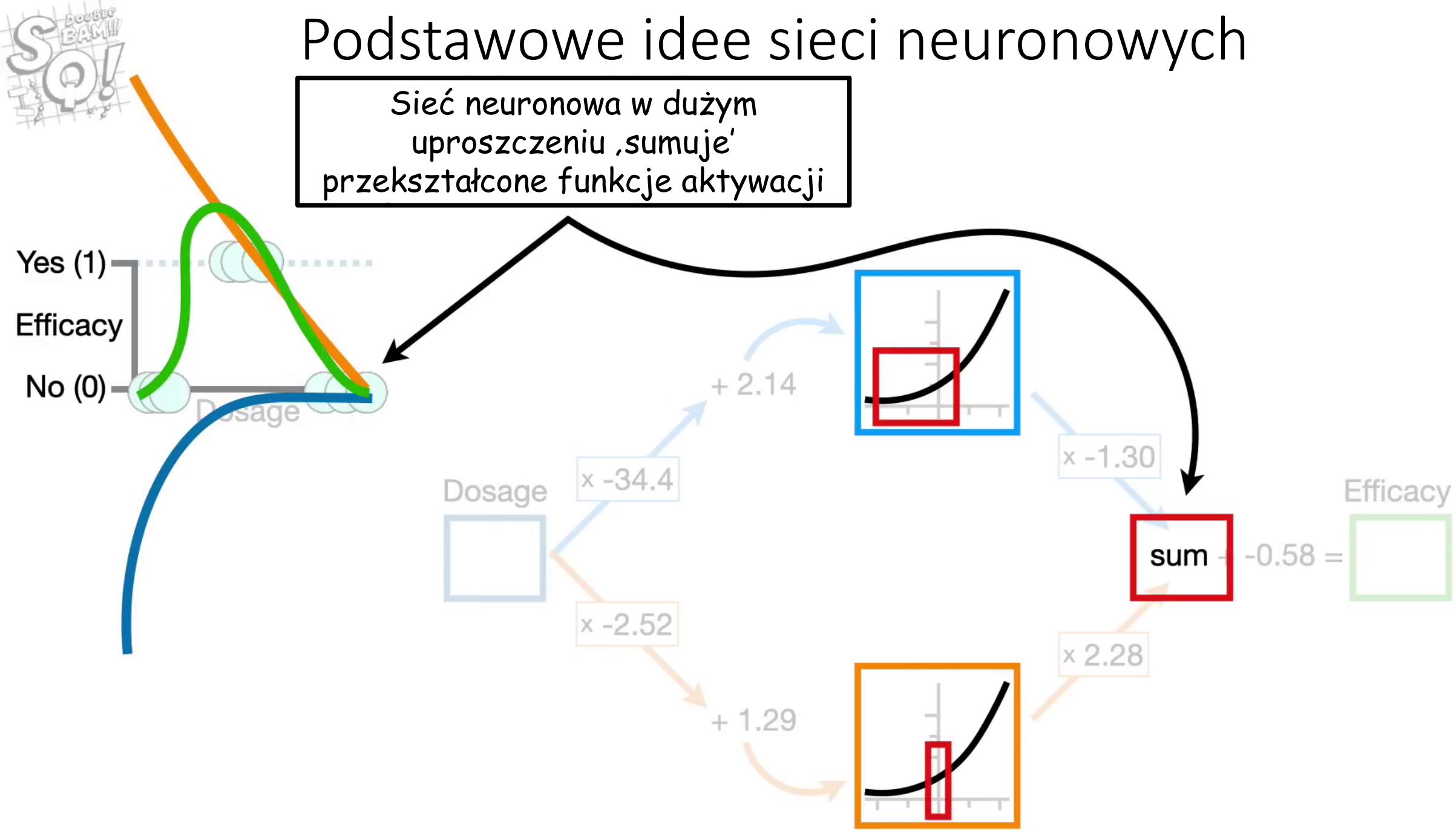
Funkcje aktywacji: skokowa (klasyfikacja), liniowa (skalowanie), softmax, Rectified Linear Unit (ReLU), sigmoid

Zainicjowane **wagi i bias** są losowe i poprawiane trakcie wstecznej propagacji (nauki sieci neuronowych na danych treningowych)

https://www.youtube.com/watch?v=CqOfi41LfDw&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=2 StatQuest with Josh Starmer

Podstawowe idee sieci neuronowych

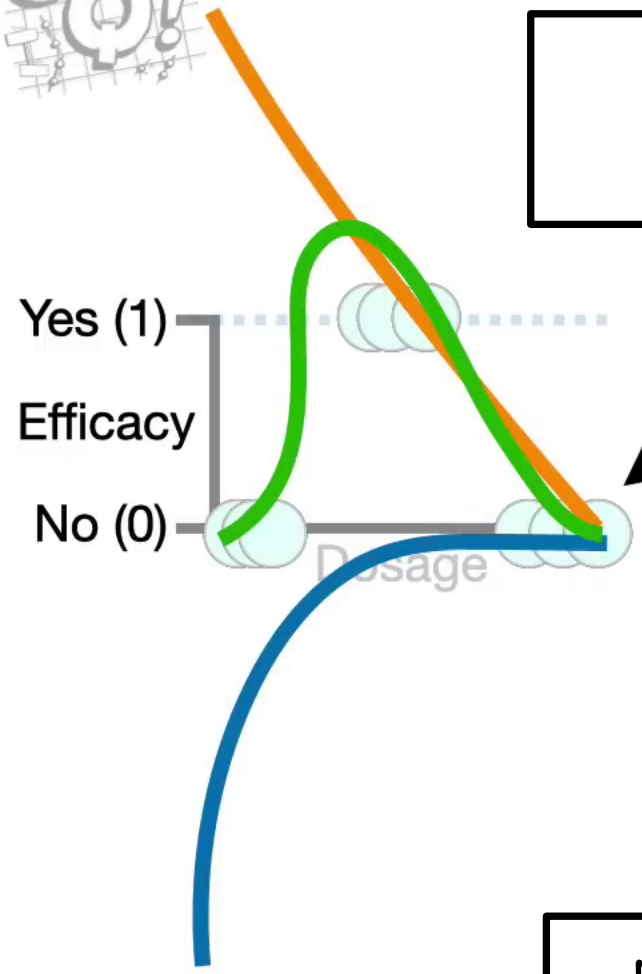
Sieć neuronowa w dużym uproszczeniu ‚sumuje' przekształcone funkcje aktywacji



https://www.youtube.com/watch?v=CqOfi41LfDw&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=2 StatQuest with Josh Starmer

Podstawowe idee sieci neuronowych

Input * *w1* + *w0* =
x funkcji aktywacji



Dawka
od 0 do 1

Dosage

0

x -34.4

0 + 2.14 = 2.14

+ 2.14

$f(x) = \log(1 + e^x)$

x -1.30

$f(\mathbf{2.14}) = \log(1 + e^{2.14}) = 2.25$

sum

-0.58 =

Efficacy

x -2.52

(**0** x -2.52) + 1.29 = 1.29

+ 1.29

$f(x) = \log(1 + e^x)$

x 2.28

$\log(1 + e^{1.29}) = 1.53$

https://www.youtube.com/watch?v=CqOfi41LfDw&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=2 StatQuest with Josh Starmer

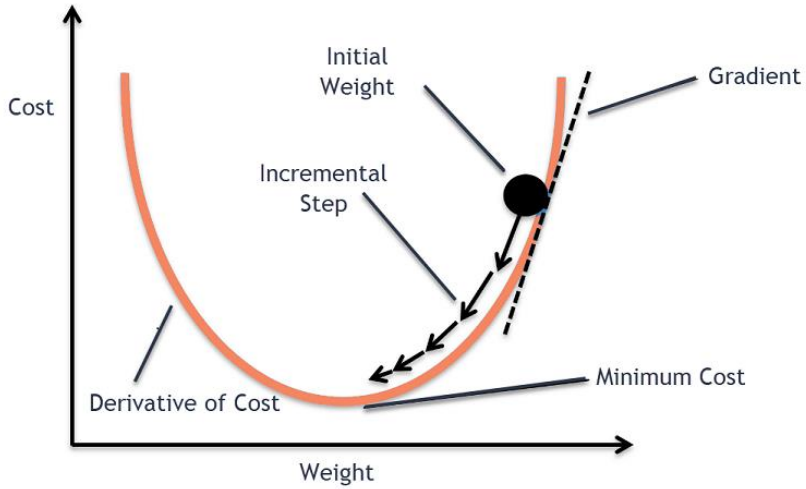
Algorytm propagacji wstecznej - *backpropagation algorithm*

Reguła łańcuchowa – pochodne funkcji złożonych

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

$$\frac{dy}{dx} = \left(\begin{array}{c}\text{Differentiate}\\ \text{outer function}\\ \\ \text{Keep the inside}\\ \text{the same}\end{array}\right)\left(\begin{array}{c}\text{Differentiate}\\ \text{inner function}\end{array}\right)$$

Metoda gradientu prostego – algorytm numeryczny do minimum lokalnego funkcji celu



Reguła łańcuchowa – pochodne funkcji złożonych

If $y = f(u)$, where $u = g(x)$

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

$$\frac{dy}{dx} = \left(\text{Differentiate outer function; Keep the inside the same}\right)\left(\text{Differentiate inner function}\right)$$

Differentiate $f(x) = \cos(2x)$

$$\frac{dy}{dx} = \left(\text{Differentiate outer function; Keep the inside the same}\right)\left(\text{Differentiate inner function}\right)$$

$$f(x) = \cos(2x)$$

$$f'(x) = -\sin(2x) \cdot 2$$

$$f'(x) = -2\sin(2x)$$

https://mathsathome.com/chain-rule-differentiation/

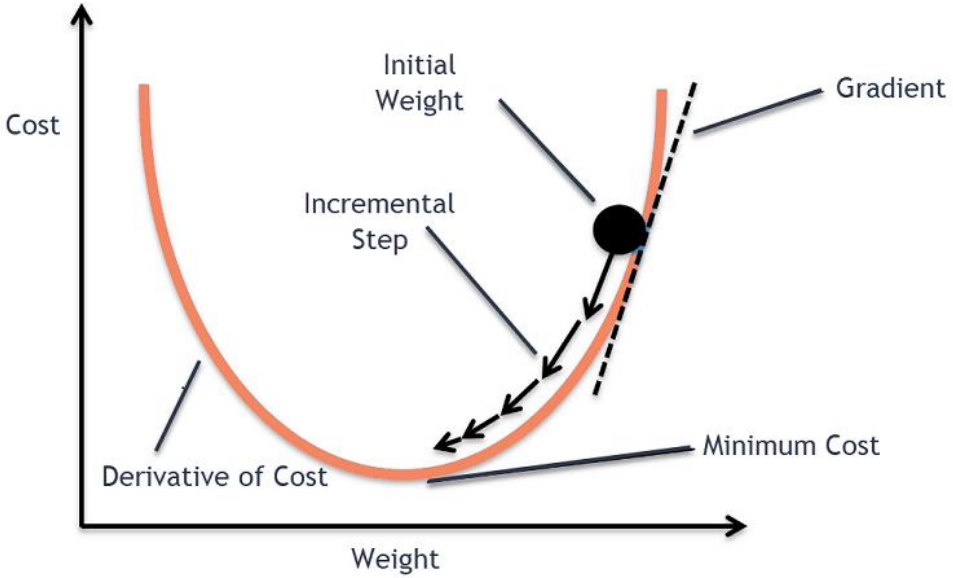
Metoda gradientu prostego – algorytm numeryczny do minimum lokalnego funkcji celu

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

Cost Function: $J(\theta_0, \theta_1) = \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$

Goal: $\underset{\theta_0,\theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

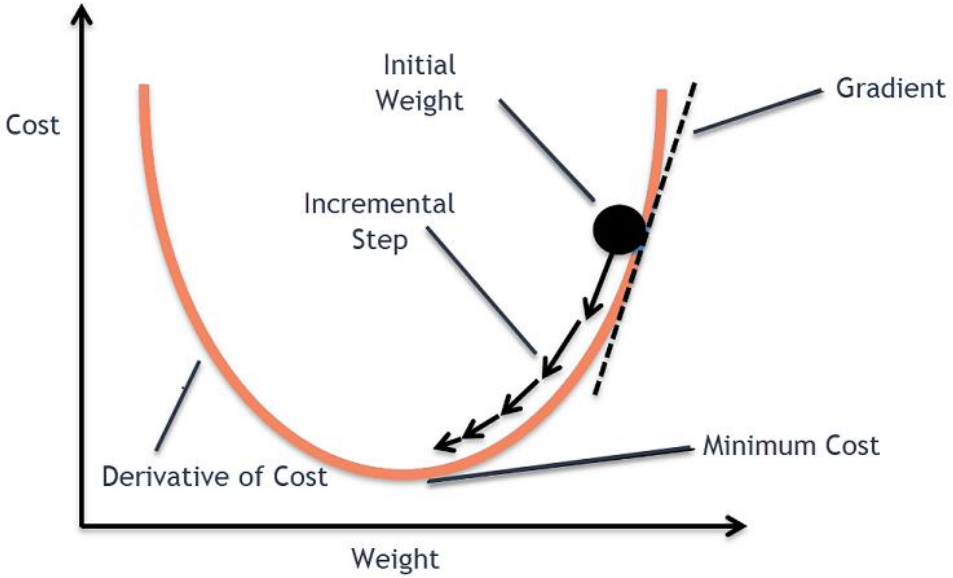


$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

https://www.analyticsvidhya.com/blog/2020/10/how-does-the-gradient-descent-algorithm-work-in-machine-learning/

Metoda gradientu prostego – algorytm numeryczny do minimum lokalnego funkcji celu



Rozważmy funkcję kwadratową $f(x) = x^2$.

Krok 1: Wybierz punkt początkowy, na przykład $x_0 = 4$.

Krok 2: Oblicz gradient funkcji w punkcie początkowym. Dla funkcji kwadratowej $f(x) = x^2$, gradient to $\frac{df}{dx} = 2x$. W punkcie $x_0 = 4$ gradient wynosi $\frac{df}{dx} = 2 \times 4 = 8$.

Krok 3: Przesuń się w kierunku przeciwnym do gradientu o pewną odległość, która jest ustalona przez współczynnik uczenia (learning rate). Załóżmy, że wybieramy współczynnik uczenia równy $\alpha = 0.1$. Zatem nowa wartość x_1 będzie wynosić:

$$x_1 = x_0 - \alpha \times \frac{df}{dx} = 4 - 0.1 \times 8 = 4 - 0.8 = 3.2$$

Krok 4: Powtarzaj te kroki, aż do spełnienia warunku stopu (np. gdy zmiana wartości funkcji między kolejnymi iteracjami jest wystarczająco mała) lub po osiągnięciu maksymalnej liczby iteracji.

Metoda gradientu prostego – algorytm numeryczny do minimum lokalnego funkcji celu

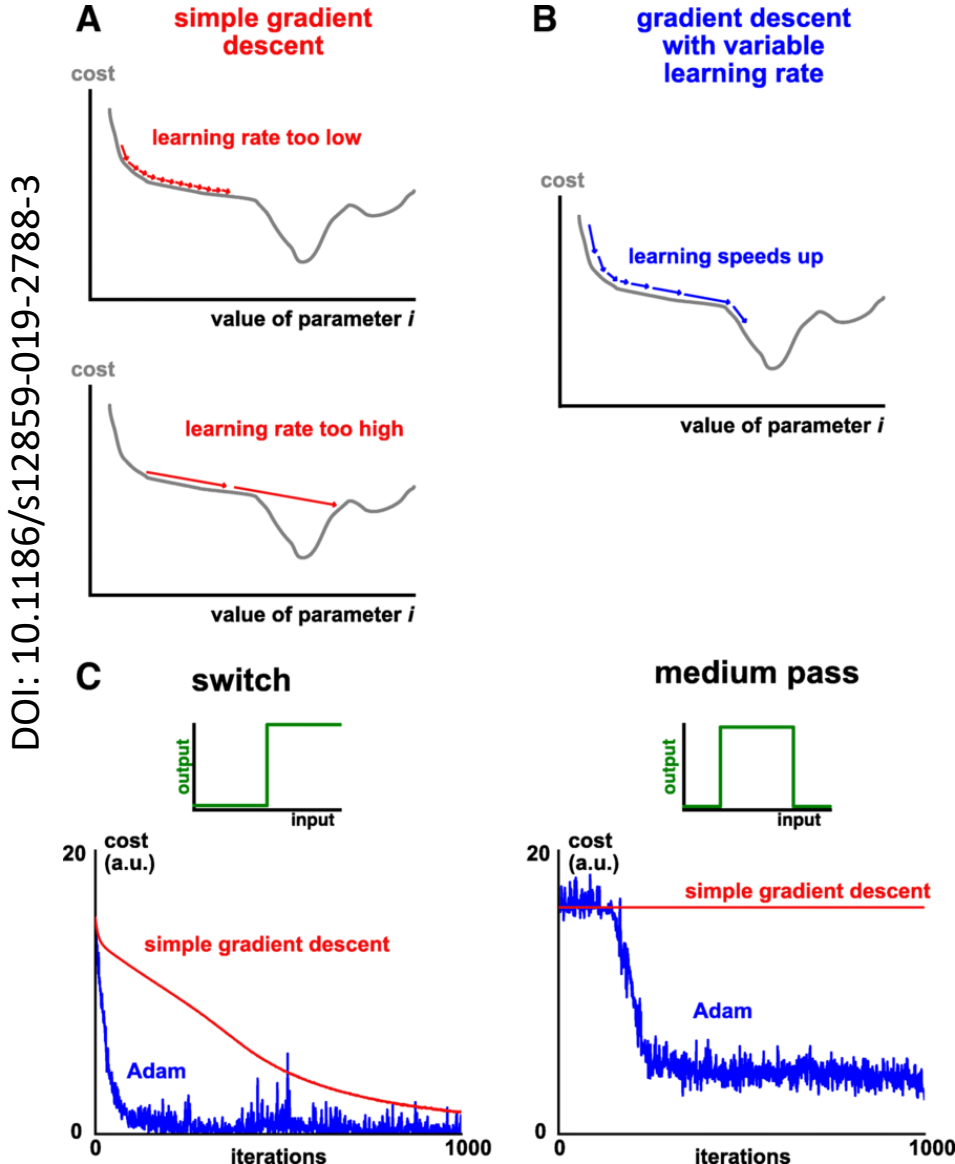
Oto kilka przykładów innych algorytmów optymalizacyjnych używanych w uczeniu maszynowym:

1.Metoda spadku gradientu ze spadkiem szybkości uczenia (learning rate decay): modyfikuje learning rate w trakcie iteracji, zmniejszając ją wraz z postępem optymalizacji.

2.Metoda momentum: dodaje "momentum" do procesu optymalizacji, czyli uwzględnia poprzednie kierunki i prędkości przemieszczania się.

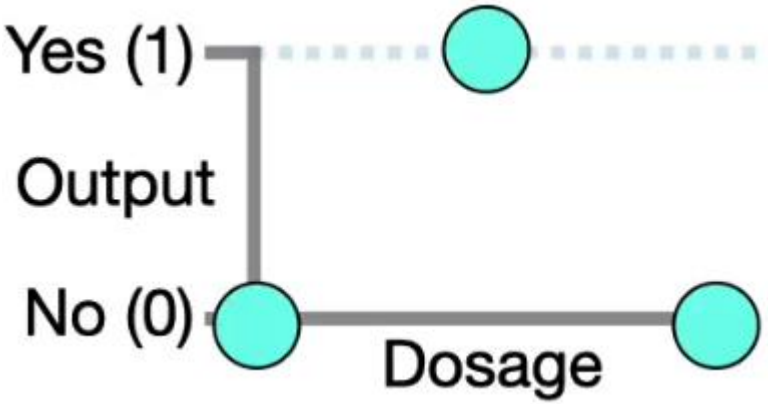
3.Algorytmy adaptacyjnego kroku uczenia (adaptive learning rate algorithms): Takie algorytmy, np. Adam, RMSprop, AdaGrad, automatycznie dostosowują wielkość kroku uczenia dla każdego parametru w zależności od jego historycznej wartości gradientu.

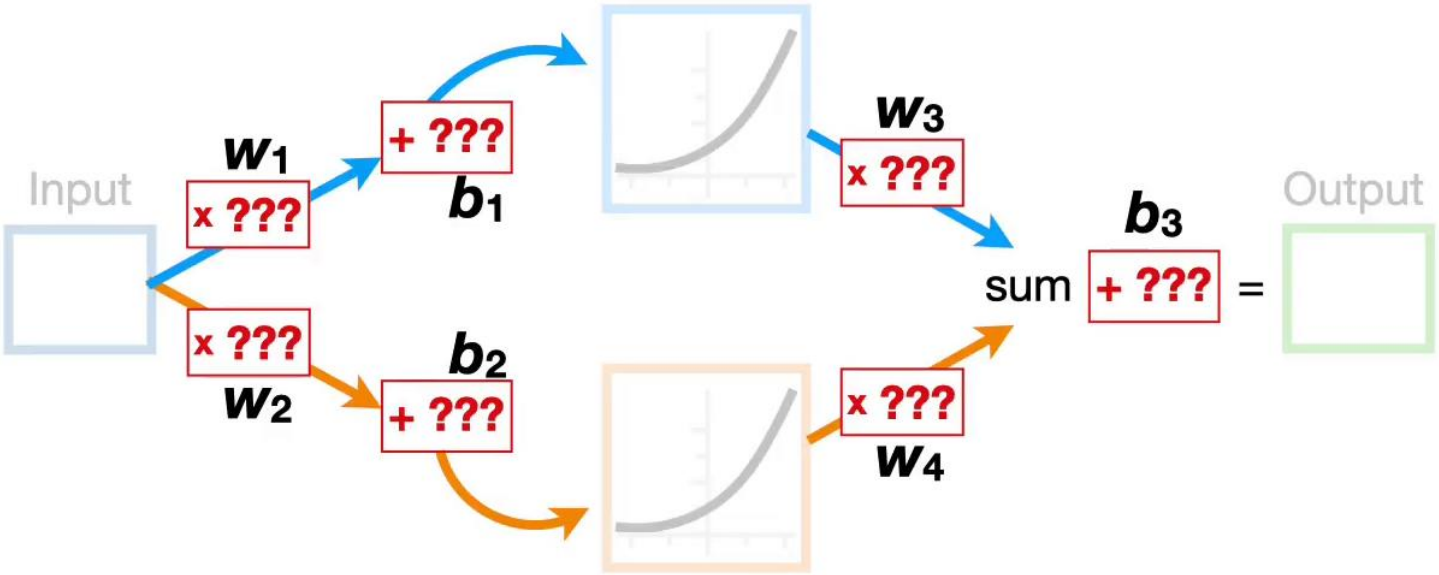
4.Algorytmy drugiego rzędu (second-order optimization algorithms): Algorytmy te, np. metoda Newtona, używają informacji z drugich pochodnych funkcji kosztu (takich jak macierz Hessego) do obliczenia kierunku optymalizacji. Mogą być bardziej skuteczne w przypadku funkcji o nieliniowym kształcie lub w przypadku wysokich wymiarów przestrzeni parametrów, ale wymagają one większych obliczeń.



DOI: 10.1186/s12859-019-2788-3

Algorytm propagacji wstecznej - *backpropagation algorithm*



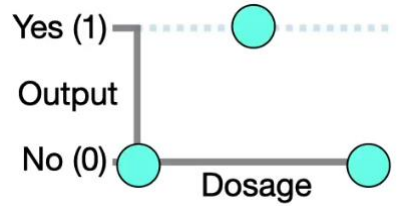


https://www.youtube.com/watch?v=GKZoOHXGcLo&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=7, Backpropagation Details Pt. 2: Going bonkers with The Chain Rule, StatQuest with Josh Starmer

Algorytm propagacji wstecznej - *backpropagation algorithm*

Jako **funkcję kosztu** *loss function* (ocena przewidywań sieci) przyjmiemy **sumę kwadratów reszt** *Sum of Squared Residuals* (**SSR**), np. dla 3 danych:

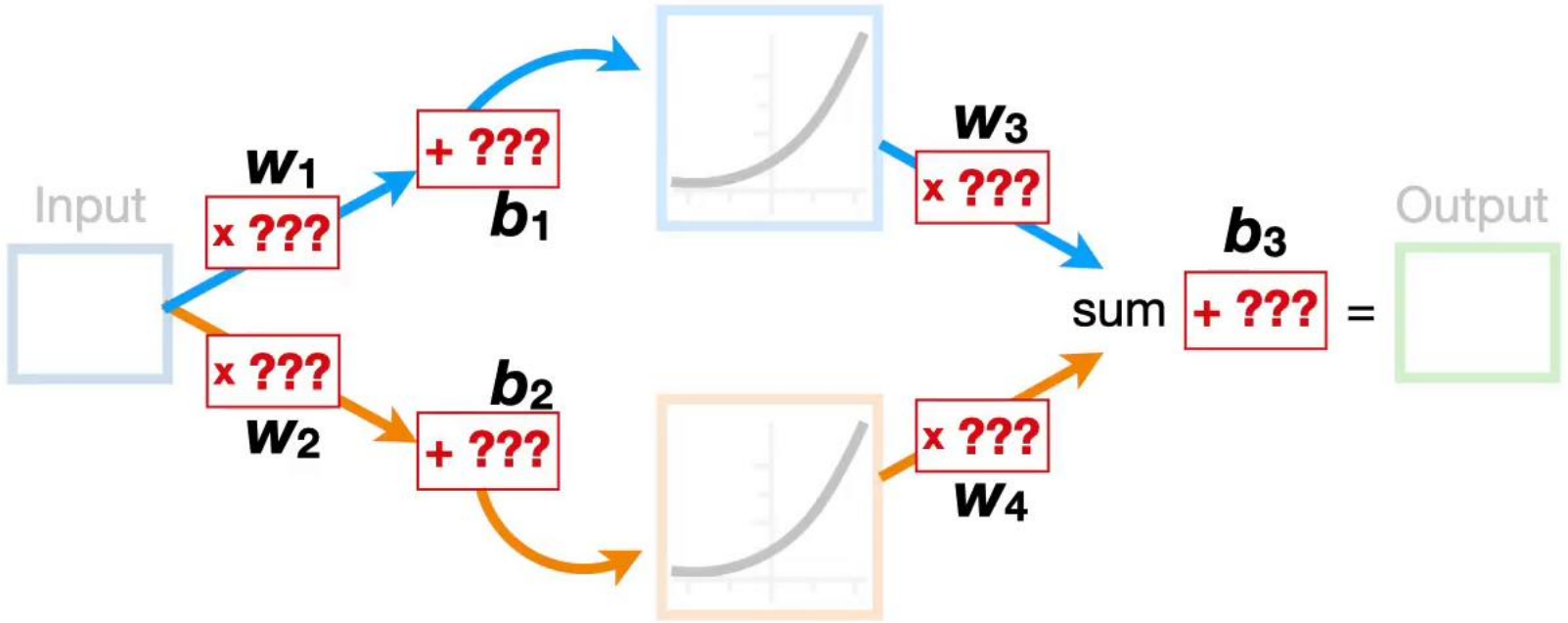
$$SSR = \sum_{i=1}^{n=3} (Observed_i - Predicted_i)^2$$



Niska wartość **funkcji kosztu** to **dobrze przewidujący model!**

Dlatego chcemy **zminimalizować funkcję kosztu**, tj. tak dobrać wagi sieci, aby SSR miało jak najmniejszą wartość.

Skorzystamy z **metody gradientu prostego**, czyli potrzebujemy **pochodnych SSR po wagach**.



$$\frac{d\,SSR}{d\,w_1}$$

$$\frac{d\,SSR}{d\,b_1}$$

$$\frac{d\,SSR}{d\,w_2}$$

$$\frac{d\,SSR}{d\,b_2}$$

itd...

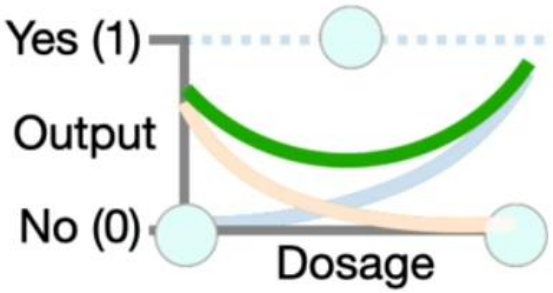
https://www.youtube.com/watch?v=GKZoOHXGcLo&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=7, Backpropagation Details Pt. 2: Going bonkers with The Chain Rule, StatQuest with Josh Starmer

Algorytm propagacji wstecznej - *backpropagation algorithm*

Szukamy pochodnej funkcji kosztu wobec wagi *w1*

$$\frac{d\,SSR}{d\,w_1} = \frac{d\,SSR}{d\,Predicted} \times \frac{d\,Predicted}{d\,y_1} \times \frac{d\,y_1}{d\,x_1} \times \frac{d\,x_1}{d\,w_1}$$

$$SSR = \sum_{i=1}^{n=3} (Observed_i - Predicted_i)^2$$

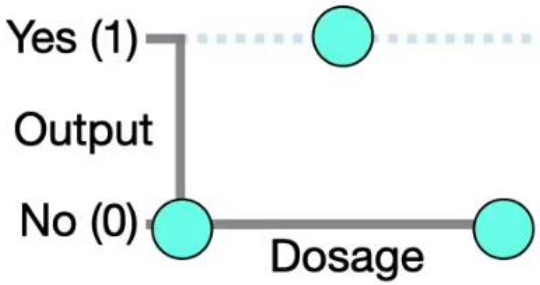


$$Predicted_i = green\ squiggle_i = y_{1,i}w_3 + y_{2,i}w_4 + b_3$$

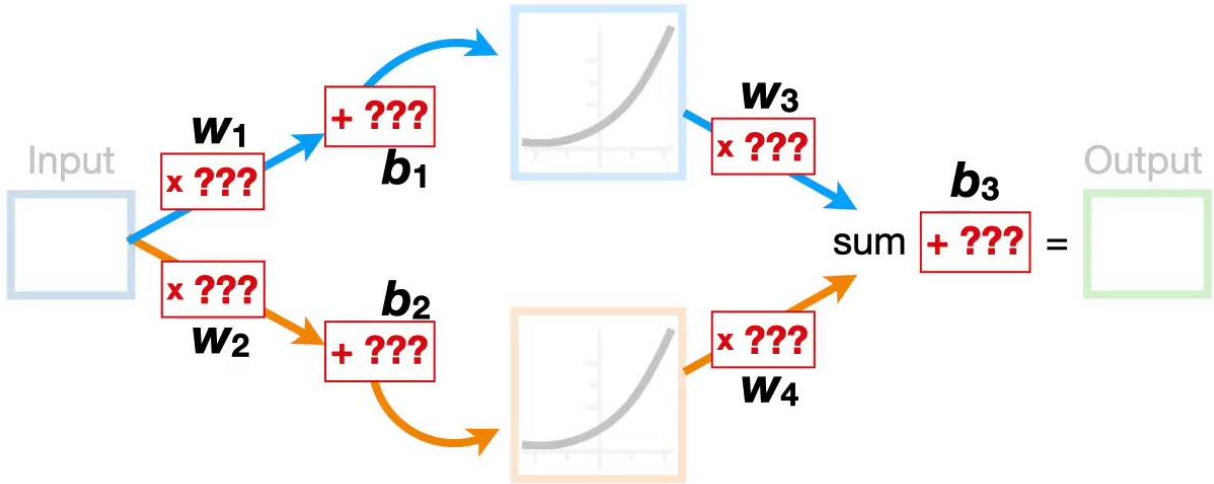
blue curve orange curve

$$y_{1,i} = f(x_{1,i}) = \log(1 + e^x)$$

Activation Function



$$x_{1,i} = Input_i \times w_1 + b_1$$



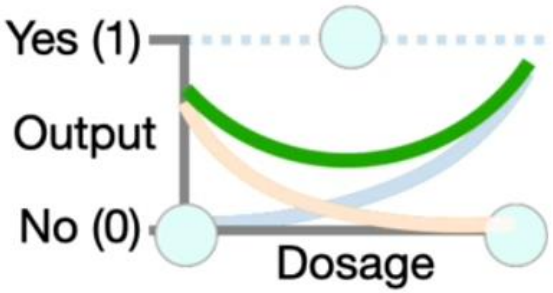
https://www.youtube.com/watch?v=GKZoOHXGcLo&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=7, Backpropagation Details Pt. 2: Going bonkers with The Chain Rule, StatQuest with Josh Starmer

Algorytm propagacji wstecznej - *backpropagation algorithm*

Po przekształceniach pochodna funkcji kosztu wobec wagi *w1*

$$\frac{d\,SSR}{d\,w_1} = \sum_{i=1}^{n=3} -2 \times (Observed_i - Predicted_i) \times w_3 \times \frac{e^x}{1+e^x} \times Input_i$$

$$SSR = \sum_{i=1}^{n=3} (Observed_i - Predicted_i)^2$$



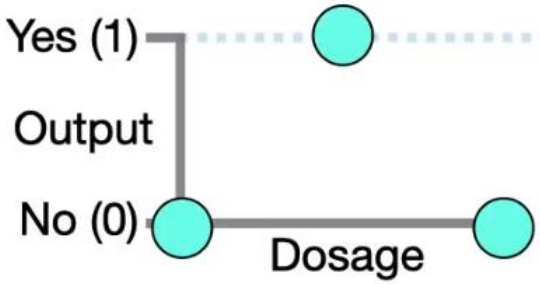
$$Predicted_i = green\ squiggle_i = y_{1,i}w_3 + y_{2,i}w_4 + b_3$$

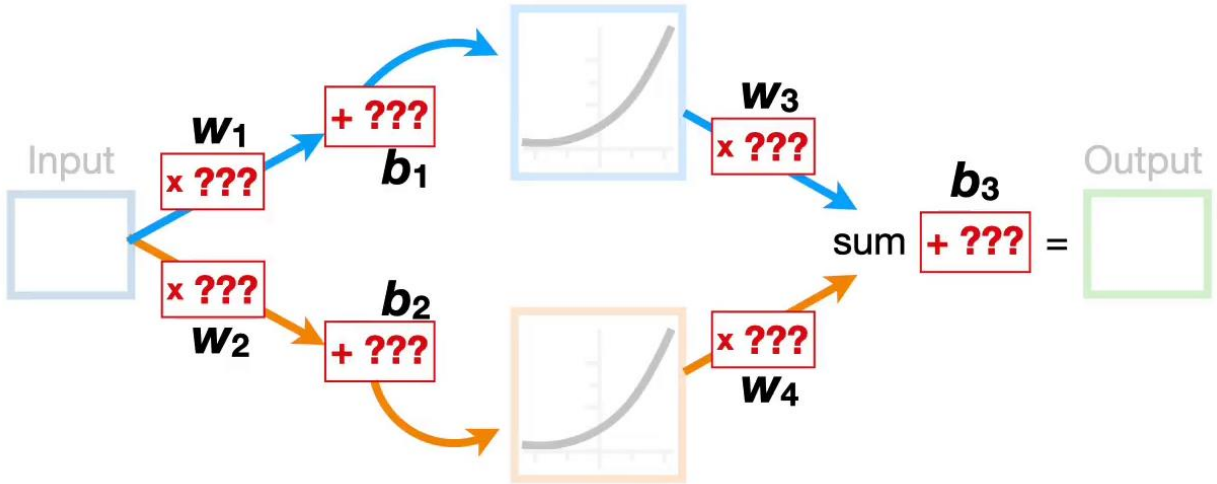
blue curve orange curve

$$y_{1,i} = f(x_{1,i}) = \log(1 + e^x)$$

Activation Function

$$x_{1,i} = Input_i \times w_1 + b_1$$





https://www.youtube.com/watch?v=GKZoOHXGcLo&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=7, Backpropagation Details Pt. 2: Going bonkers with The Chain Rule, StatQuest with Josh Starmer

Algorytm propagacji wstecznej - *backpropagation algorithm*

$$\frac{d\,SSR}{d\,w_1} = \sum_{i=1}^{n=3} -2 \times (Observed_i - Predicted_i) \times w_3 \times \frac{e^x}{1+e^x} \times Input_i$$

$$\frac{d\,SSR}{d\,b_1} = \sum_{i=1}^{n=3} -2 \times (Observed_i - Predicted_i) \times w_3 \times \frac{e^x}{1+e^x} \times 1$$

$$\frac{d\,SSR}{d\,w_2} = \sum_{i=1}^{n=3} -2 \times (Observed_i - Predicted_i) \times w_4 \times \frac{e^x}{1+e^x} \times Input_i$$

$$\frac{d\,SSR}{d\,b_2} = \sum_{i=1}^{n=3} -2 \times (Observed_i - Predicted_i) \times w_4 \times \frac{e^x}{1+e^x} \times 1$$

$$\frac{d\,SSR}{d\,w_3} = \sum_{i=1}^{n=3} -2 \times (Observed_i - Predicted_i) \times y_{1,i}$$

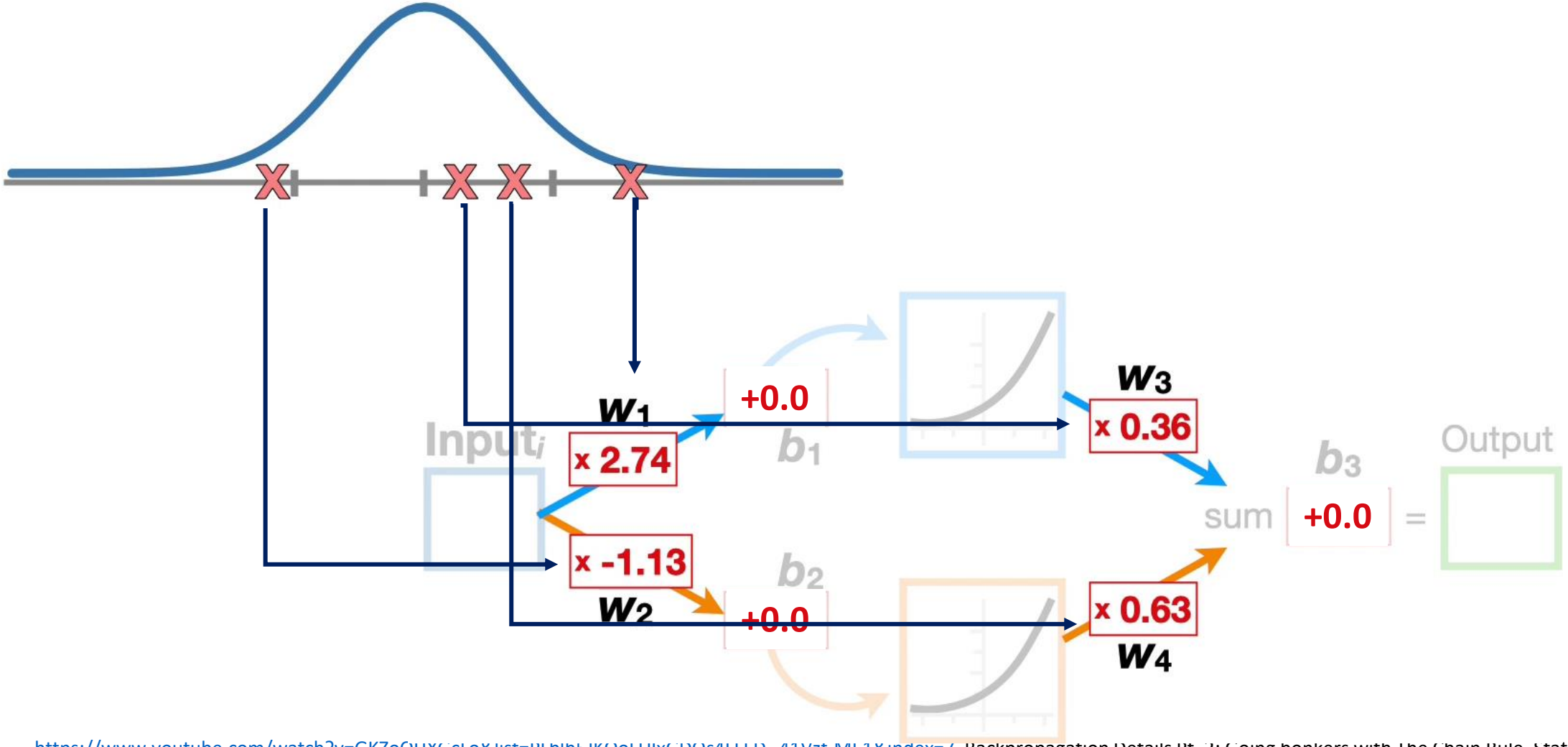
$$\frac{d\,SSR}{d\,w_4} = \sum_{i=1}^{n=3} -2 \times (Observed_i - Predicted_i) \times y_{2,i}$$

https://www.youtube.com/watch?v=GKZoOHXGcLo&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=7, Backpropagation Details Pt. 2: Going bonkers with The Chain Rule, StatQuest with Josh Starmer

Algorytm propagacji wstecznej - *backpropagation algorithm*

Inicjalizacja: Losowe inicjalizowanie wag sieci neuronowej, np. korzystając z rozkładu normalnego (średnia 0, odchylenie standardowe 1), a wartości bias 0.

Wagi są parametrami, które definiują zachowanie sieci neuronowej, więc **sieć neuronowa** jest uznawana **za model parametryczny.**



https://www.youtube.com/watch?v=GKZoOHXGcLo&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=7, Backpropagation Details Pt. 2: Going bonkers with The Chain Rule, StatQuest with Josh Starmer

Algorytm propagacji wstecznej - *backpropagation algorithm*

Inicjalizacja: Losowe inicjalizowanie wag sieci neuronowej, np. korzystając z rozkładu normalnego (średnia 0, odchylenie standardowe 1), a wartości bias 0.

Popularne techniki inicjalizacji wag:

1. **Losowa inicjalizacja**: Wagi są inicjalizowane losowo, na przykład z rozkładu normalnego.

2. **Inicjalizacja Xaviera / Glorot**: Inicjalizacja wag W_{ij} jest dokonywana zgodnie z zaleceniami zaproponowanymi przez Xaviera Glorota: U rozkładem normalnym, fan_{in} rozmiar poprzedniej warstwy (liczbę kolumn), fan_{out} oznacza rozmiar bieżącej warstwy.

$$W_{ij} \sim U\left[-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}\right]$$

3. **Inicjalizacja He**: Podobnie jak inicjalizacja Xaviera, ale **dostosowana do funkcji aktywacji ReLU** (Rectified Linear Unit), aby lepiej radzić sobie z problemem zanikającego gradientu.

4. **Inicjalizacja Orthogonalna**: Wagi są inicjalizowane jako macierz ortogonalna, co może pomóc w uniknięciu zjawiska korelacji między wagami. (Jak pomnożymy macierz przez jej otrzymamy macierz jednostkową). Wiersze reprezentują wagi wychodzące z neuronów w jednej warstwie, a kolumny reprezentują wagi wchodzące do neuronów w kolejnej warstwie,

Algorytm propagacji wstecznej - *backpropagation algorithm*

Inicjalizacja: Losowe inicjalizowanie wag sieci neuronowej, np. korzystając z rozkładu normalnego (średnia 0, odchylenie standardowe 1), a wartości bias 0.

Zły dobór inicjalizacji wag w sieciach neuronowych może prowadzić do kilku istotnych konsekwencji:

1. **Zanikający lub eksplodujący gradient**: Nieprawidłowo zainicjalizowane wagi mogą prowadzić do zjawiska zanikającego lub eksplodującego gradientu, co może znacząco utrudnić proces uczenia się poprzez propagację wsteczną.

2. **Wolniejszy lub zatrzymany postęp treningu**: Nieoptymalne wagi mogą prowadzić do wolniejszego postępu w treningu lub nawet całkowicie zatrzymać proces uczenia się, co może prowadzić do słabszej jakości modelu.

3. **Nadmierna czułość na początkowe dane**: Jeśli wagi są zbyt małe lub zbyt duże, model może stać się nadmiernie czuły na dane treningowe, co może prowadzić do przeuczenia lub niedouczenia.

4. **Niestabilność modelu**: Nieprawidłowo zainicjalizowane wagi mogą prowadzić do niestabilnego zachowania modelu podczas treningu, co może skutkować nieregularnymi wynikami lub trudnościami w zbieżności algorytmu uczenia się.

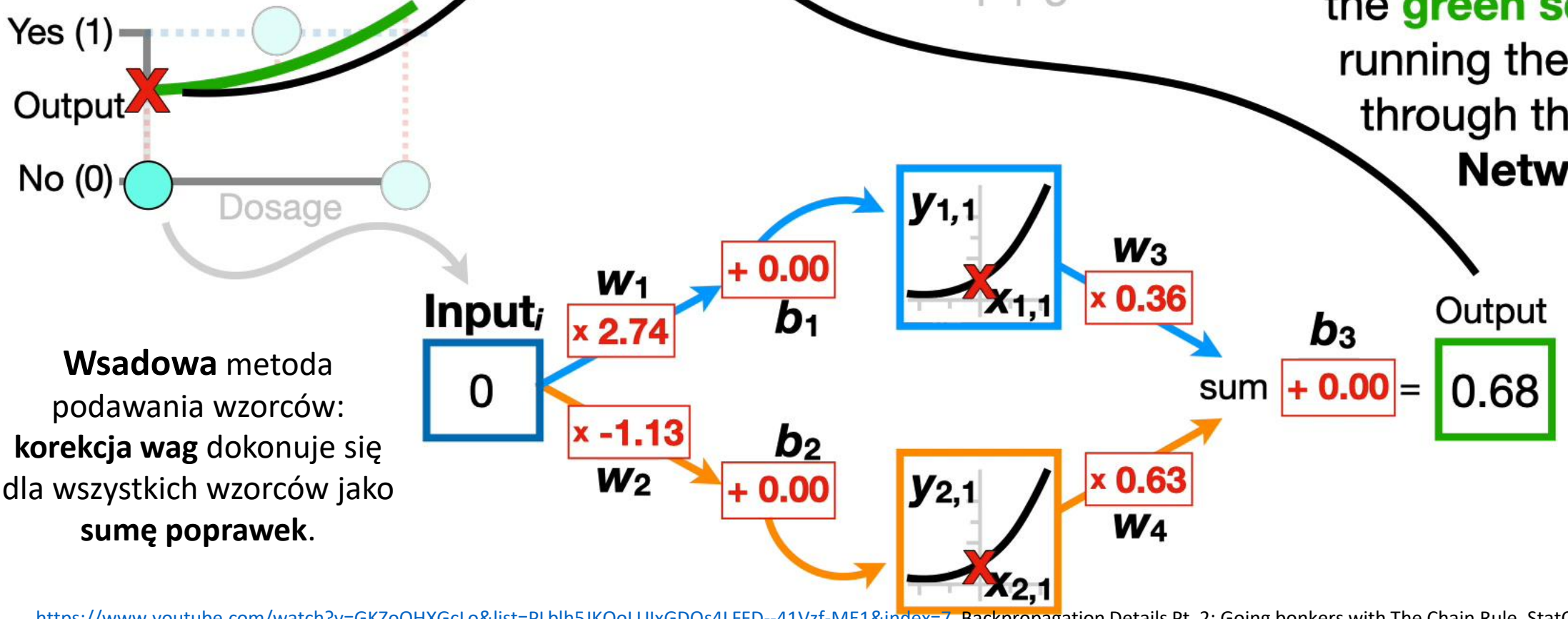
Algorytm propagacji wstecznej - *backpropagation algorithm*

$$\frac{d\,SSR}{d\,w_1} = -2 \times (0 - 0.68) \times w_3 \times \frac{e^x}{1 + e^x} \times Input_1$$

$$+ -2 \times (1 - Predicted_2) \times w_3 \times \frac{e^x}{1 + e^x} \times Input_2$$

$$+ -2 \times (0 - Predicted_3) \times w_3 \times \frac{e^x}{1 + e^x} \times Input_3$$

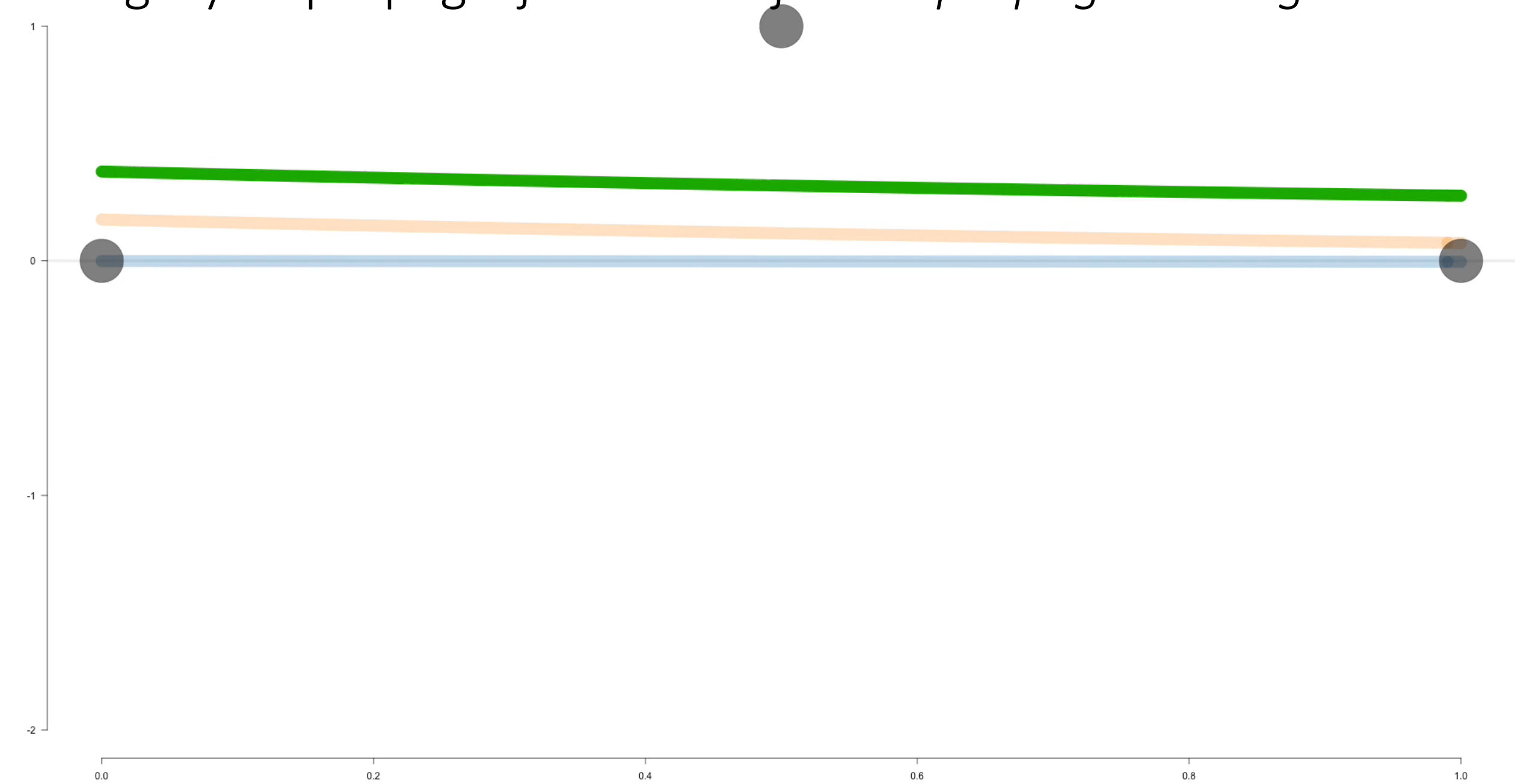
Remember, we get the **Predicted** values on the **green squiggle** by running the **Dosages** through the **Neural Network.**



Wsadowa metoda podawania wzorców: **korekcja wag** dokonuje się dla wszystkich wzorców jako **sumę poprawek**.

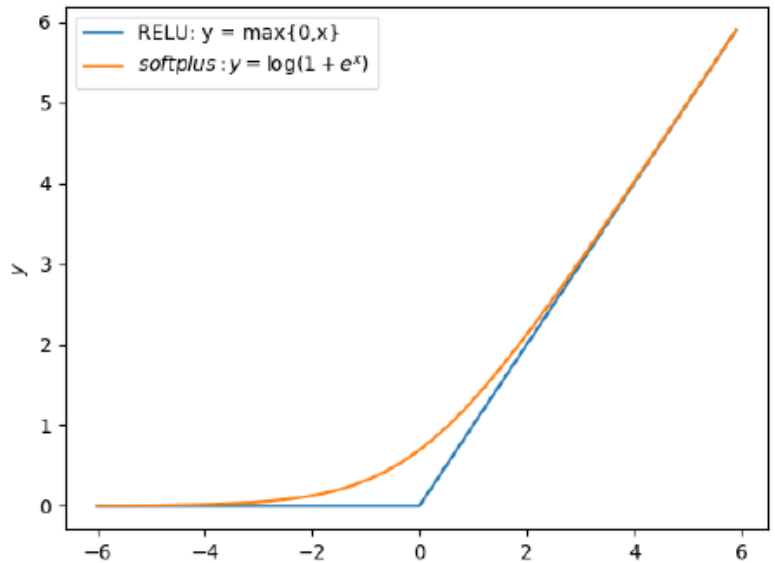
https://www.youtube.com/watch?v=GKZoOHXGcLo&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=7, Backpropagation Details Pt. 2: Going bonkers with The Chain Rule, StatQuest with Josh Starmer

Algorytm propagacji wstecznej - *backpropagation algorithm*



https://www.youtube.com/watch?v=GKZoOHXGcLo&list=PLblh5JKOoLUIxGDQs4LFFD--41Vzf-ME1&index=7, Backpropagation Details Pt. 2: Going bonkers with The Chain Rule, StatQuest with Josh Starmer

Funkcja (ang. Rectified Linear Unit) RELU



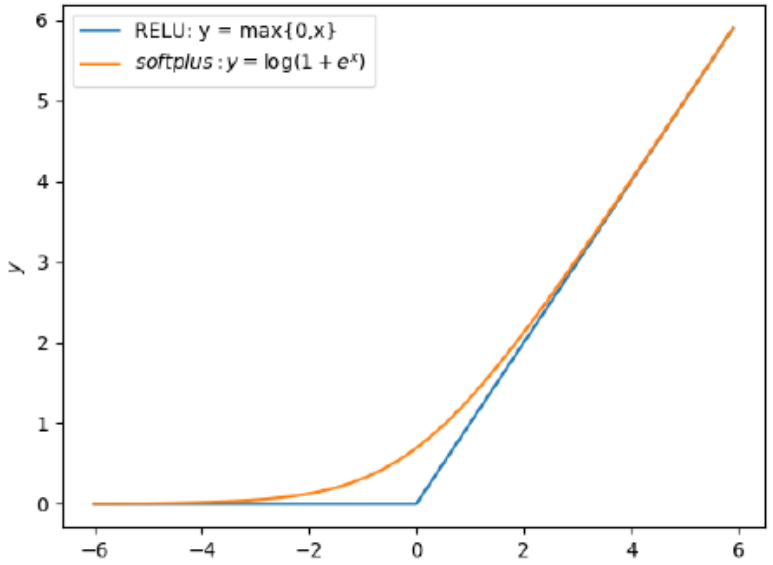
- Funkcja RELU: $y = max\{x, 0\}$
- Funkcja RELU można aproksymować funkcją ciągłą *softplus*:

$$\zeta = \log(1 + e^x)$$

- Pochodną funkcji *softplus* jest funkcja *sigmoidalna*:

$$\frac{d}{dx}\zeta(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Funkcja (ang. Rectified Linear Unit) RELU



- Funkcja RELU: $y = max\{x, 0\}$

W większości bibliotek pochodna funkcji ReLU jest zazwyczaj zaimplementowana jako:

$$f'(x) = \begin{cases} 0 & \text{dla } x < 0 \\ 1 & \text{dla } x > 0 \\ 0 \text{ lub } 1 & \text{dla } x = 0 \end{cases}$$

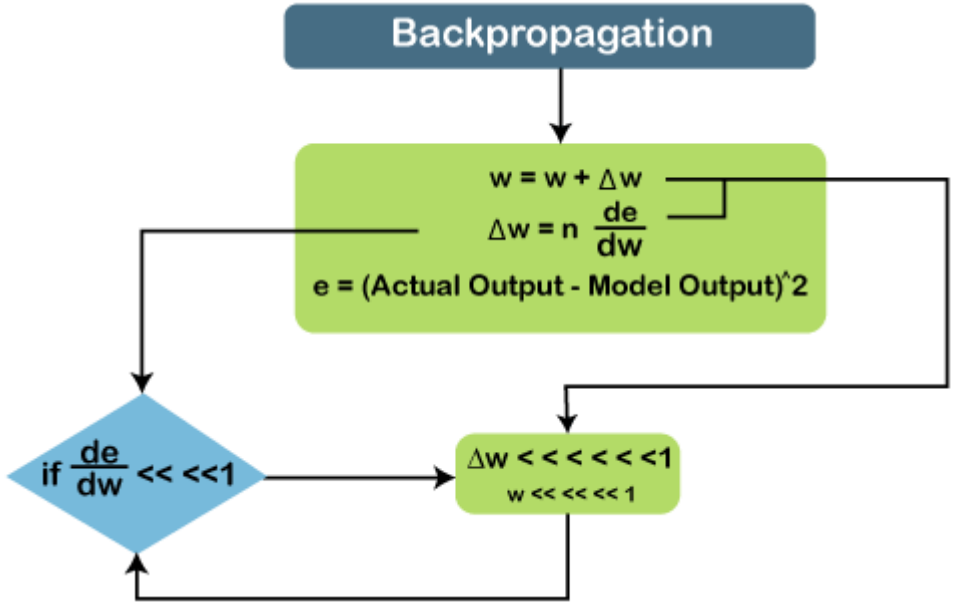
Często dokonuje się różnych prób z ustawieniami 0 lub 1 dla *x=0*, aby zbadać, które zachowanie pochodnej działa lepiej dla danego problemu. Ustawienie pochodnej na 1 może umożliwić neuronowi nadal uczenie się nawet w przypadku zerowego sygnału wejściowego.

Najczęściej stosowane funkcje aktywacji

rodzaj	równanie	pochodna	zastosowanie
skokowa	$\mathrm{f(s)} = \begin{cases} 1 & s > 0 \\ 0 & s \leq 0 \end{cases}$	–	klasyfikacja
liniowa	$f(s) = s$	$f'(s) = 1$	skalowanie
unipolarna	$f(s) = \frac{1}{1+e^{-s}}$	$f'(s) = f(s)(1 - f(s))$	mod. nielin.
bipolarna	$f(s) = \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}}$	$f'(s) = (1 - f(s) \cdot f(s))$	mod. nielin.
arc tang.	$f(s) = atan(s)$	$f'(s) = \frac{1}{1+s^2}$	mod. nielin.

Zanikający lub eksplodujący **gradient**

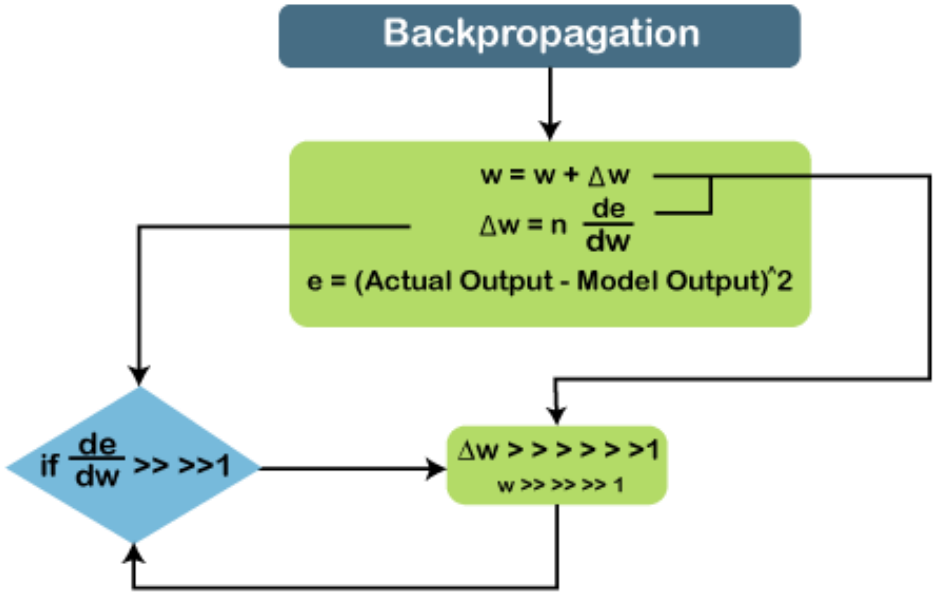
- Podczas **propagacji wstecznej gradienty stają się coraz mniejsze gdy przechodzimy coraz głębiej w sieci.**
- Neurony w wcześniejszych warstwach uczą się bardzo wolno w porównaniu do neuronów w późniejszych warstwach.
- **Spadek wartości gradientu jest proporcjonalny do głębokości sieci.**
- Problem **znikającego gradientu** występuje głównie przy użyciu **funkcji sigmoidalnej i tangens hiperboliczny**.
- Możemy **uniknąć** tego problemu, **stosując funkcje aktywacji ReLU** podczas treningu głębokich sieci neuronowych.



https://medium.com/@sjakki/vanishing-and-exploding-gradients-in-neural-networks-5b3ee108a568

Zanikający lub **eksplodujący gradient**

- **Eksplodujące gradienty** osiągają **bardzo duże wartości podczas propagacji wstecznej**. Powodem tego jest przykładowo zbyt duża wartość początkowa wag.
- W skrajnych przypadkach prowadzi to do **utraty stabilności treningu** i uniemożliwienia osiągnięcia odpowiedniej wydajności sieci.
- Aby **uniknąć** problemu eksplodującego gradientu, stosuje się różne techniki, takie jak **ograniczenie wartości gradientu**, **użycie stabilnych metod optymalizacyjnych**, jak również **odpowiednie skalowanie i inicjalizacja wag**.



https://medium.com/@sjakki/vanishing-and-exploding-gradients-in-neural-networks-5b3ee108a568

Regularyzacja L2

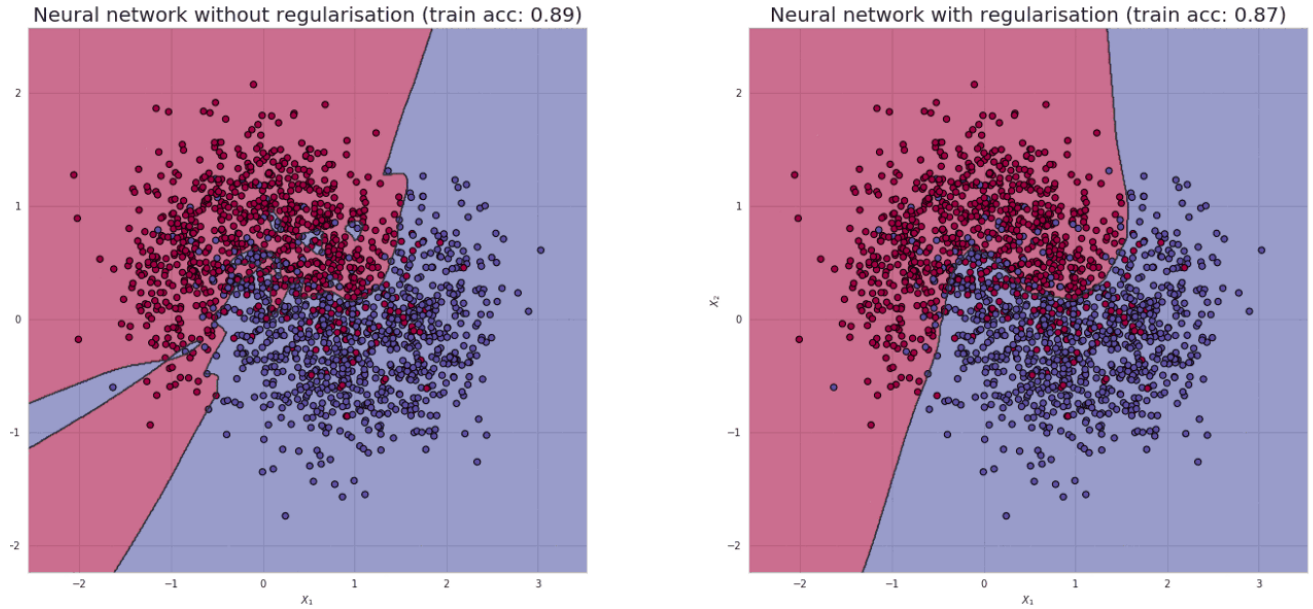
Regularyzacja L2 jest techniką stosowaną w celu zapobiegania przeuczeniu się modelu poprzez karanie dużych wartości wag w sieci neuronowej. Działa poprzez dodanie kary do funkcji kosztu proporcjonalnej do kwadratu wartości wag.

W przypadku sieci neuronowej, koszt z regularyzacją L2 może być zapisany jako:

$$J_{\text{regularized}} = J_{\text{original}} + \frac{\lambda}{2} \sum_i \sum_j W_{ij}^2$$

Gdzie:

- J_{original} to oryginalna funkcja kosztu,
- λ to parametr regularyzacji L2 (reg_lambda),
- W_{ij} to waga połączenia między neuronami i i j.



Parametr `**reg_lambda**` kontroluje, jak bardzo wagi sieci są penalizowane za ich wielkość.

https://medium.com/analytics-vidhya/regularization-understanding-l1-and-l2-regularization-for-deep-learning-a7b9e4a409bf

Ile neuronów nieliniowych powinna mieć sieć?

Optymalizacja architektury sieci jest często wykonywana przez **eksperymentowanie z różnymi konfiguracjami sieci** podczas procesu uczenia i walidacji: **walidacja krzyżowa** lub wykorzystanie zbioru testowego.

Istnieją **techniki automatycznej optymalizacji architektury** sieci:

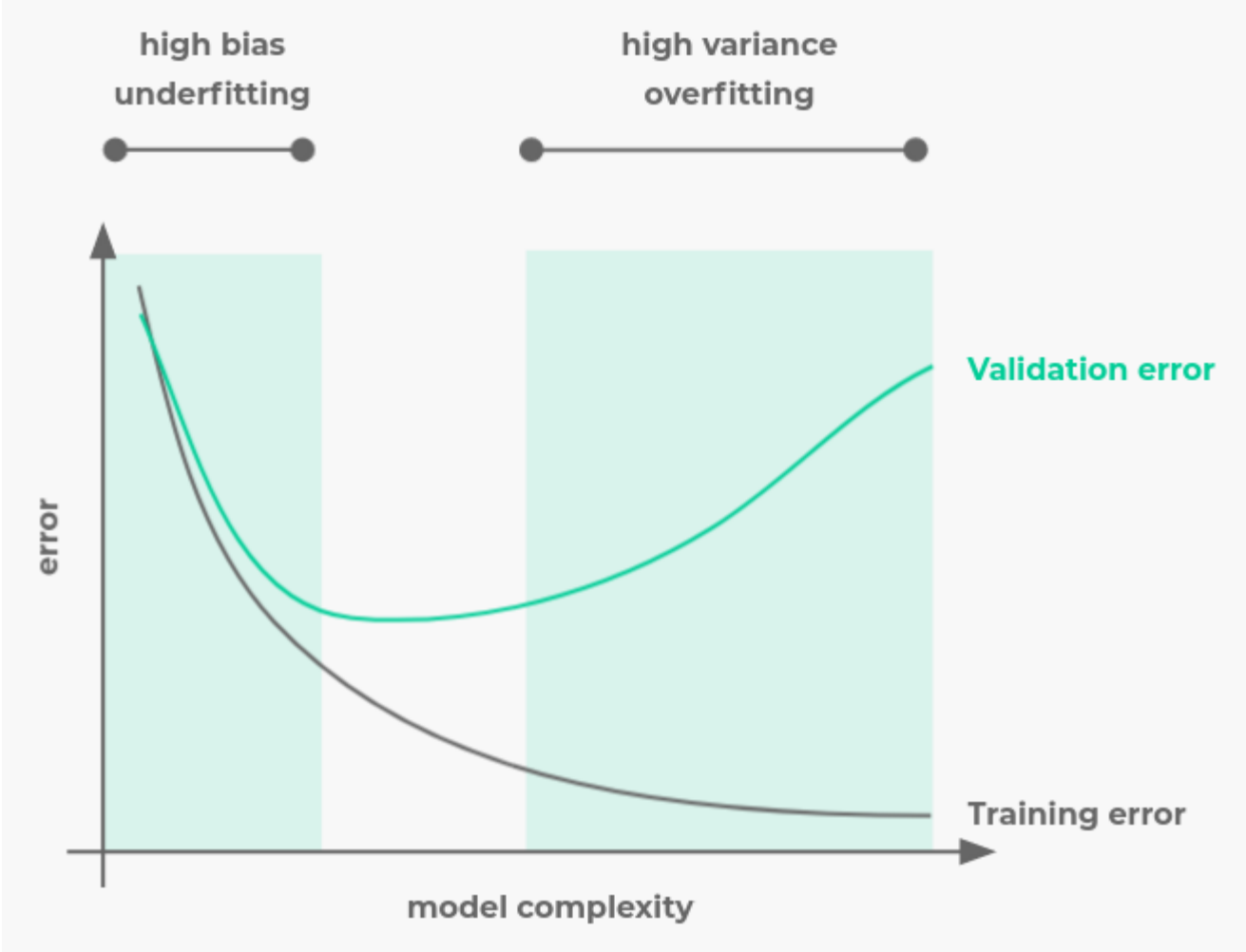
1.Grid Search: przetestowaniu wszystkich możliwych kombinacji hiperparametrów zdefiniowanych w określonym zakresie (kosztowna obliczeniowo).

2.Random Search: losowo wybiera zestawy hiperparametrów do przetestowania.

3.Metody ewolucyjne: inspirowane biologiczną ewolucją do przeszukiwania przestrzeni hiperparametrów.

4.Bayesian Optimization: Jest to metodologia oparta na probabilistycznym modelowaniu

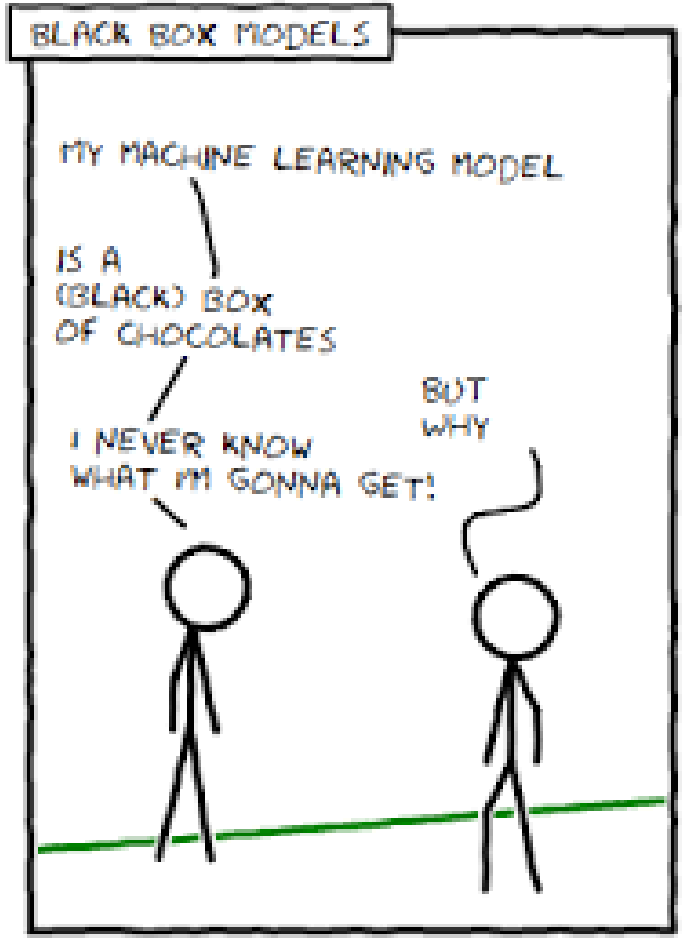
5.Automatyczne strojenie hiperparametrów (AutoML): To podejście wykorzystuje zaawansowane techniki do automatycznego strojenia hiperparametrów.



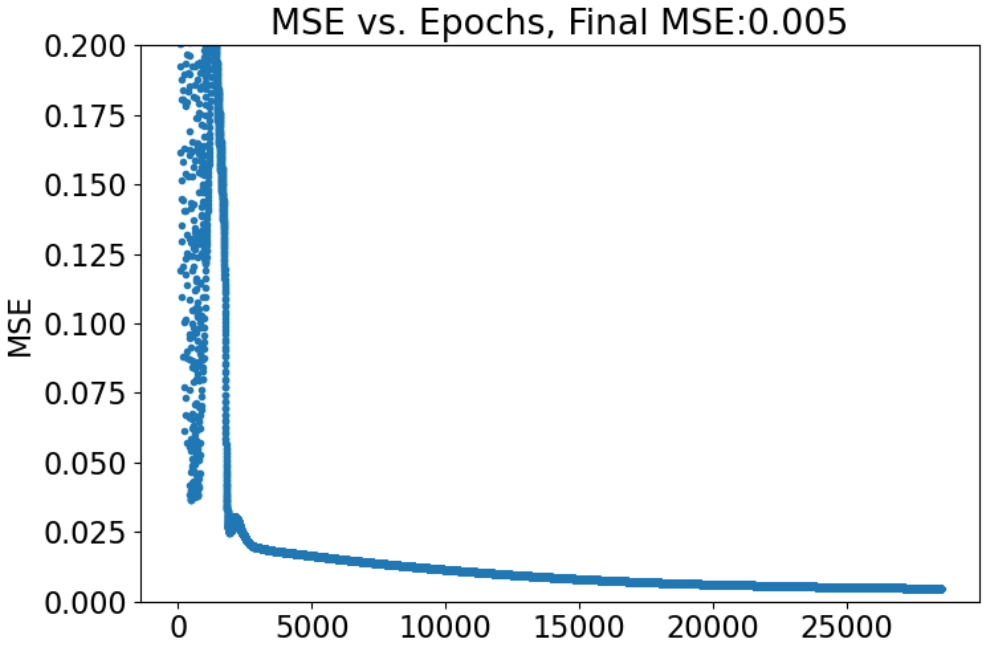
Implementacja i trening prostego modelu sieci neuronowej dla regresji

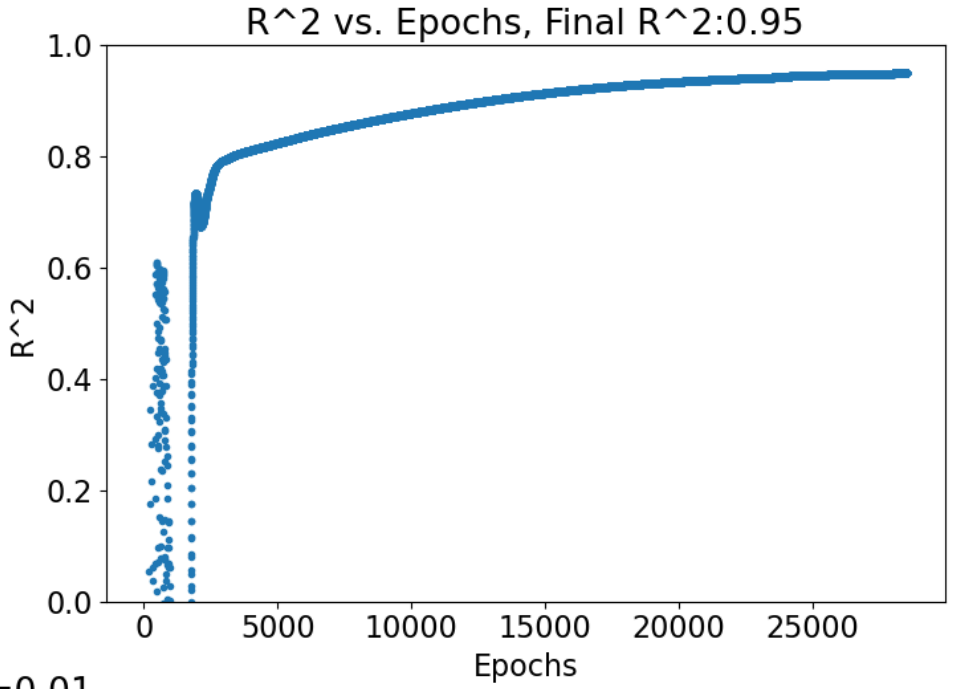
Celem tego projektu jest **zrozumienie i praktyczna implementacja prostego modelu sieci neuronowej do zadania regresji**.

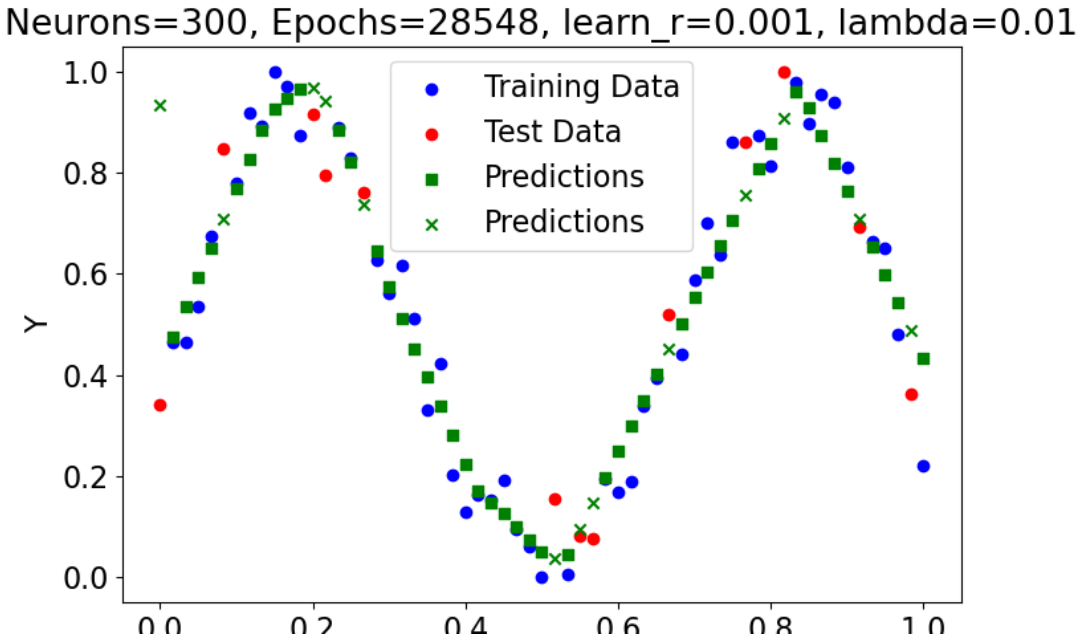
1. **Implementacja klasy NeuralNetworkRegression:**
    1. Zaimplementuj klasę NeuralNetworkRegression, która będzie zawierała metody do
        - inicjalizacji wag,
        - propagacji sygnału w przód
        - i wstecz oraz
        - trenowania sieci neuronowej.
    2. Wykorzystaj funkcję aktywacji ReLU oraz jej pochodną.
    3. Dodaj historię wartości MSE i R^2 podczas treningu.
2. **Przygotowanie danych:**
    1. Wczytaj dane z pliku tekstowego *Dane/daneXX.txt* (XX od 1 do 16).
    2. Podziel dane na cechy (X) i etykiety (y).
    3. Wykonaj normalizację danych: np. za pomocą *MinMaxScaler*.
    4. Podziel dane na zbiór treningowy i testowy za pomocą funkcji *train_test_split.*
3. **Trenowanie sieci neuronowej:**
    1. Zainicjalizuj obiekt klasy *NeuralNetworkRegression* z odpowiednimi parametrami.
    2. Trenuj sieć neuronową na zbiorze treningowym przez określoną liczbę epok.
    3. Zapisz metryki MSE i R^2 do oceny jakości predykcji podczas treningu.
4. **Ocena wyników:**
    1. Po zakończeniu treningu, oblicz wartości MSE i R^2 dla danych treningowych i testowych.
    2. Wyświetl wykresy zmiany wartości MSE i R^2 w kolejnych epokach treningu.
    3. Wygeneruj wykres z punktami danych treningowych i testowych oraz przewidywaniami sieci neuronowej.



Przykładowa strona raportu dla *Dane/dane3.txt*:







Funkcja aktywacji: ***ReLU***
Pochodna f.akt.: *0 dla x<0, 1 dla x>0,* ***0.5 dla x=0***

Inicjalizacja **wag**: **rozkład normalny** (średnia 0, odchylenie 1)
Inicjalizacja **bias'ów:** wartość **0**

MSE, r^2score, liczba neuronów, epoch, współczynnik uczenia i lambda podano w nagłówkach wykresów.

Kodujemy

Implementacja i trening prostego modelu sieci neuronowej dla regresji

public/mmajew/MIW/07/

00_neural_network_step_by_step.py

Kodujemy

Implementacja i trening prostego modelu sieci neuronowej dla regresji

1. Propagacja sygnału w przód (forward propagation):

- W procesie propagacji sygnału w przód, dla danej próbki wejściowej X, obliczamy aktywacje neuronów w warstwie ukrytej (H) oraz w warstwie wyjściowej ($\hat{Y}$).
- Aktywacje w warstwie ukrytej obliczane są jako iloczyn skalarny między wejściami X a wagami W_{input_hidden}, do których dodajemy biasy b_{hidden}. Następnie stosujemy funkcję aktywacji ReLU, co daje nam aktywacje H.
- Aktywacje w warstwie wyjściowej obliczane są w podobny sposób, z tym że jako wejścia bierzemy aktywacje z warstwy ukrytej, a nie dane wejściowe. Otrzymane aktywacje O są naszymi przewidywanymi wartościami.
- Matematycznie można to zapisać jako:

$$H = \text{ReLU}(X \cdot W_{input\_hidden} + b_{hidden})$$
$$\hat{Y} = \text{ReLU}(H \cdot W_{hidden\_output} + b_{output})$$

Gdzie:

- X to macierz danych wejściowych o wymiarach (n, m), gdzie n to liczba próbek, a m to liczba cech,
- W_{input_hidden} to macierz wag między warstwą wejściową a ukrytą o wymiarach (m, k), gdzie k to liczba neuronów w warstwie ukrytej,
- b_{hidden} to wektor biasów dla warstwy ukrytej o długości k,
- H to macierz aktywacji warstwy ukrytej o wymiarach (n, k),
- W_{hidden_output} to macierz wag między warstwą ukrytą a wyjściową o wymiarach $(k, 1)$,
- b_{output} to wektor biasów dla warstwy wyjściowej o długości 1,
- $\hat{Y}$ to przewidywane wartości regresji dla danych wejściowych.

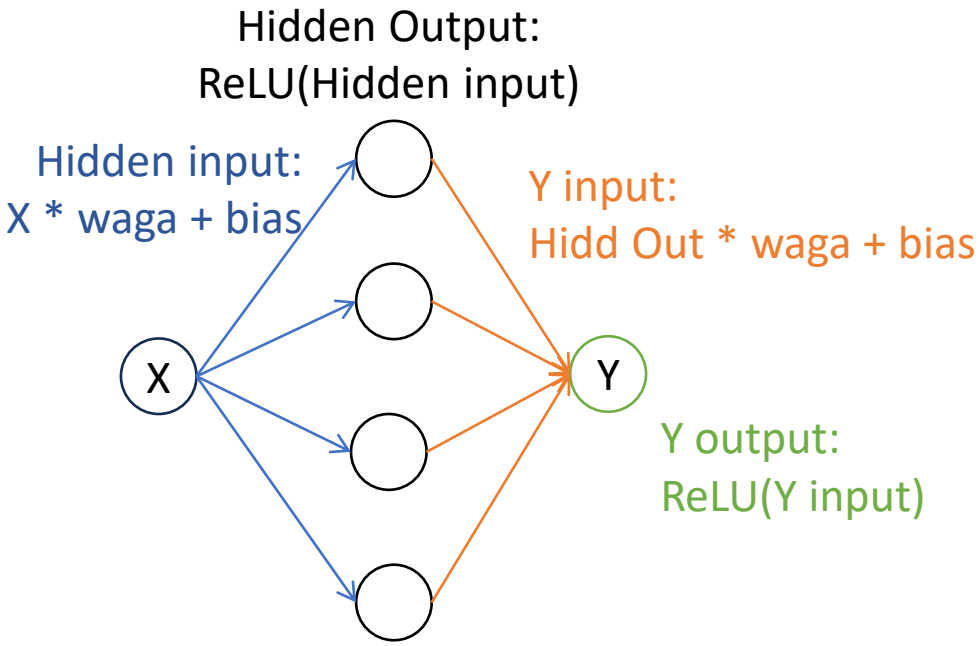
Kodujemy

Implementacja i trening prostego modelu sieci neuronowej dla regresji

1. Propagacja sygnału w przód (forward propagation):

- W procesie propagacji sygnału w przód, dla danej próbki wejściowej X, obliczamy aktywacje neuronów w warstwie ukrytej (H) oraz w warstwie wyjściowej ($\hat{Y}$).
- Aktywacje w warstwie ukrytej obliczane są jako iloczyn skalarny między wejściami X a wagami W_{input_hidden}, do których dodajemy biasy b_{hidden}. Następnie stosujemy funkcję aktywacji ReLU, co daje nam aktywacje H.
- Aktywacje w warstwie wyjściowej obliczane są w podobny sposób, z tym że jako wejścia bierzemy aktywacje z warstwy ukrytej, a nie dane wejściowe. Otrzymane aktywacje O są naszymi przewidywanymi wartościami.
- Matematycznie można to zapisać jako:

$$H = \text{ReLU}(X \cdot W_{input\_hidden} + b_{hidden})$$
$$\hat{Y} = \text{ReLU}(H \cdot W_{hidden\_output} + b_{output})$$



Kodujemy

Implementacja i trening prostego modelu sieci neuronowej dla regresji

2. Propagacja wsteczna błędu (backpropagation):

- W procesie propagacji wstecznej błędu, obliczamy gradienty funkcji kosztu względem wag w sieci neuronowej, aby zaktualizować wagi i minimalizować błąd.
- Najpierw obliczamy błąd w warstwie wyjściowej jako różnicę między prawdziwymi etykietami Y a przewidywanymi wartościami $\hat{Y}$.
- Następnie obliczamy gradient funkcji kosztu (np. MSE) względem wag między warstwą ukrytą a wyjściową W_{hidden_output}, korzystając z reguły łańcuchowej.
- Podobnie obliczamy gradient błędu w warstwie ukrytej, aby zaktualizować wagi między warstwą wejściową a ukrytą W_{input_hidden}.
- Wagi są aktualizowane w kierunku przeciwnym do gradientu, z uwzględnieniem współczynnika uczenia oraz opcjonalnej regularyzacji (np. L2).

- Matematycznie można to zapisać jako:

$$\begin{aligned}
\text{Output error} &= Y - \hat{Y} \\
\text{Gradient for } W_{hidden_output} &= H^T \cdot \text{Output error} \\
\text{Hidden error} &= \text{Output error} \cdot (W_{hidden_output})^T \cdot \text{ReLU de} \\
\text{Gradient for } W_{input_hidden} &= X^T \cdot \text{Hidden error}
\end{aligned}$$

Gdzie:

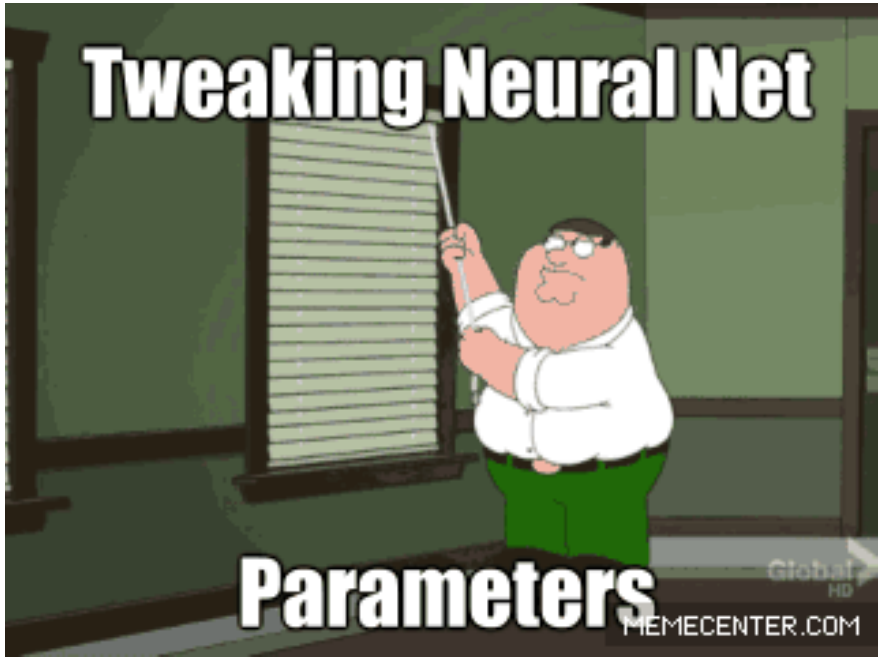
- Y to prawdziwe etykiety,
- $\text{ReLU derivative}(H)$ to pochodna funkcji aktywacji ReLU dla aktywacji warstwy ukrytej H.

Projekt 4.a (5 pkt)

Szkic: public/mmajew/MIW/07/01_neural_network_simple.py

Dla danych *Dane/daneXX.txt* dla *XX={4,7,9,15,16}* dobierz parametry sieci neuronowej:

- funkcję aktywacji i jej pochodną, (NIE wolno zastosować ReLU we wszystkich zestawach XX danych)
- sposób inicjalizacji wag i bias'ów,
- parametr uczenia
- parametru regularyzacji L2 *reg_lambda*



tak, aby uzyskać
(priorytetem jest wysoka poprawność przewidywań sieci):

- metrykę r^2 score jak najbliżej 1
  (powyżej 0.9 jest dobrze, ale czasem nie da rady więcej jak 0.5 uzyskać),
- metrykę MSE jak najbliższą 0.00,

dla jak najmniejszej liczby:

- epoch i
- neuronów.

Wynik Twoich studiów przedstaw w formie raportu w pdf: załącz wykresy optymalnych rozwiązań z wyszczególnieniem w/w parametrów (przykładowa strona raportu na poprzednim slajdzie).
Uwaga: dla każdego zestawu danych będzie inna sieć.