ZPR PWr
Zintegrowany Program Rozwoju Politechniki Wrocławskiej

SIECI NEURONOWE – laboratorium

Ćwiczenie 2 – sieć wielowarstwowa uczona metodą propagacji wstecznej (realizacja ćwiczenia na laboratorium3, laboratorium4, laboratorium5)

Liczba punktów: 60

Celem ćwiczenia jest zapoznanie się z siecią wielowarstwową, uczeniem sieci za pomocą algorytmu propagacji wstecznej w wersji klasycznej (minimalizacja błędu) średniokwadratowego oraz wpływem parametrów odgrywających istotną rolę w uczeniu sieci z propagacją wsteczną.
Na realizację opisanego niżej ćwiczenia przeznaczone są trzy laboratoria, czyli 3 tygodnie czasu. Pierwsze zajęcia przeznaczone są na implementację architektury sieci, drugie implementację metody uczenia, trzecie przeprowadzenie wstępnych badań.

Uwaga: Raport z ćwiczenia nie powinien zawierać części teoretycznej a jedynie dokumentować przeprowadzone eksperymenty. Każdy eksperyment powinien być powtórzony 10- krotnie

UWAGA: w tym ćwiczeniu można używać pakietu do mnożenia macierzy (numpy) jednak zbudowanie architektury sieci jak również implementacja metody uczenia musi być wykonana samodzielnie.

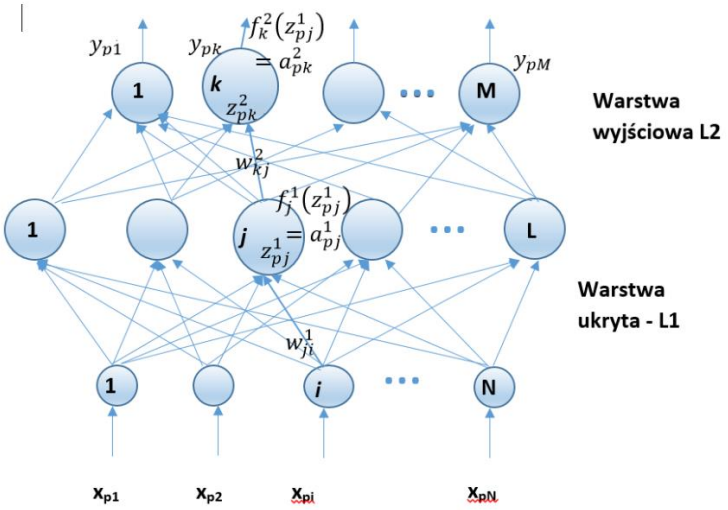
Fundusze Europejskie Wiedza Edukacja Rozwój
Politechnika Wrocławska
Unia Europejska Europejski Fundusz Społeczny

Laboratorium3 – Implementacja architektury sieci wielowarstwowej

Cel: Należy zaimplementować architekturę wielowarstwowego perceptronu do rozpoznawania cyfr ze zbioru MNIST i fazę przesłania wzorca w przód.

Wprowadzenie teoretyczne

Wielowarstwowy perceptron (MLP – MultiLayer Perceptron) jest uogólnieniem sieci, którymi zajmowaliśmy się w poprzednim ćwiczeniu. BPN jest siecią warstwową z przesyłaniem sygnału do przodu, w pełni połączoną między sąsiednimi warstwami. A więc nie istnieją połączenia ze sprzężeniami zwrotnymi i połączeniami, które omijając jakąś warstwę przechodzą do innej. Dozwolonych jest więcej niż jedna warstwa ukryta i wówczas taka sieć jest przykładem sieci głębokiej. W ćwiczeniu będziemy budować sieć z jedną warstwą ukrytą jednak w części teoretycznej będziemy odwoływać się do uogólnionego przypadku z większą liczbą warstw ukrytych.



Rys. 1. Wielowarstwowy perceptron z jedną warstwą ukrytą – wersja omawiana na wykładzie .

Odnosząc się do Rys. 1. przypomnijmy, że wektor **x** oznacza wzorce podawane do sieci neuronowej (dolny indeks p – oznacza p-ty wzorzec) **z** oznacza całkowite pobudzenie neuronu, f zaś oznacza funkcję aktywacji, której wynikiem działania jest aktywacja **a** neuronu. Zawsze górny indeks w oznaczeniach będzie się odnosił do numeru warstwy. Sieć uczona jest w trybie nadzorowanym, a to oznacza, że
mamy dany zbiór uczący D w postaci podanej poniżej:

$$D = \{\langle \mathbf{x}_1, \mathbf{y}_1 \rangle, \langle \mathbf{x}_2, \mathbf{y}_2 \rangle \dots \langle \mathbf{x}_P, \mathbf{y}_P \rangle\}$$

Dana jest sieć neuronowa z parametrami θ, która realizuje funkcję $f_\theta(\mathbf{x})$.

Naszym zadaniem jest wyuczenie parametrów θ (wag i biasów), takich, że $\forall i \in [1, N]: f_\theta(\mathbf{x}_i) = \mathbf{y}_i$
W tym ćwiczeniu nasze wektory **x** to obrazy cyfr ze zbioru MNIST a odpowiadające im wektory **y** na wyjściu, to etykiety cyfr.

Obliczenie $f_\theta(\mathbf{x})$ odbywa się w fazie przesłania p-tego wektora wejsciowego w przód aż do obliczenia wyjścia (faza forward), jak na Rys 1., i polega na obliczeniu dla każdego neuronu w danej warstwie najpierw jego całkowitego pobudzenia z, czyli dla warstwy pierwszej mielibyśmy:

$$z_{pj}^1 = \sum_{i=1}^{N} w_{ji}^1 x_{pi}$$

Natomiast wyjście z tej warstwy byłoby równe:

$$a_{pj}^{1} = f_j^1(z_{pj}^1)$$

W warstwie drugiej całkowite pobudzenie liczy się w identyczny sposób tylko tym razem wejściem jest wyjście z poprzedniej warstwy

$$z_{pk}^{2} = \sum_{j=0}^{L} w_{kj}^{2}\, a_{pj}^{1}$$

A wyjściem z tej warstwy jest wyjście z sieci **y** a dla pojedynczego *k*-tego neuronu y_{pk}

$$y_{pk} = f_k^2(z_{pk}^2)$$

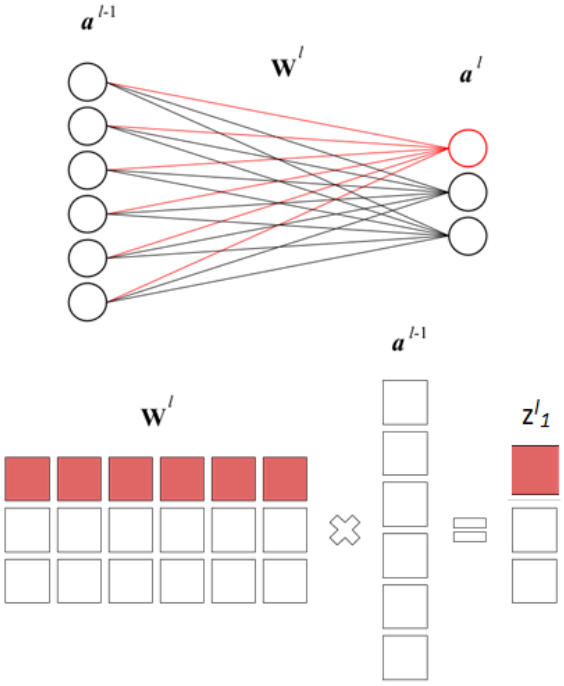
Jednak znacznie lepiej (i szybciej jeśli chodzi o obliczenia) jest używać notacji wektorowej.

Odnosząc się do dowolnej warstwy, aktywacje neuronów w warstwie *l* pamiętane są w wektorze kolumnowym $\mathbf{a}^l$ natomiast wagi na połączeniach między warstwą l oraz l+1 są pamiętane w macierzy $\mathbf{W}^l$ a biasy w wektorze kolumnowym $\mathbf{b}^l$
Dla fazy przesłania sygnału do przodu dla warstwy l, w której funkcja aktywacji oznaczona jest jako f, aktywację możemy wyrazić jako:

$$\mathbf{a}^l = f(\mathbf{W}^l\mathbf{z}^{l-1} + \mathbf{b}^l)$$

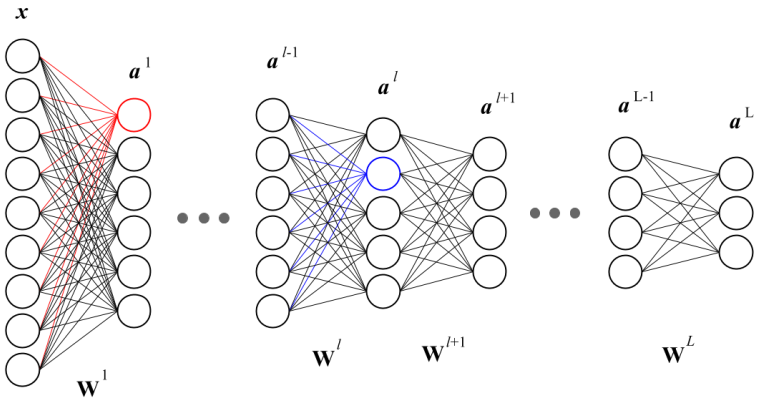
Powyższe mnożenie macierzy możemy zwizualizować jak na Rys.2., gdzie wprowadzony jest nowy wektor $\mathbf{z}^l$ oznaczający całkowite pobudzenie neuronu a następnie do tego wektora stosujemy funkcję aktywacji f, wykonując przekształcenie przez tę funkcję każdej składowej wektora.

$$\mathbf{a}^l = f(\mathbf{z}^l)$$



Rys. 2. Wizualizacja obliczania całkowitego pobudzenia jako operacji macierzowej; z^l_1 jest całkowitym pobudzeniem dla pierwszego neuronu w warstwie *l*

Cała sieć, o dowolnej liczbie warstw pokazana jest na Rys.3. Wejściem jest wektor **x**, a wyjściem z warstwy *l* jest wektor $\mathbf{a}^l$. Połączenia prowadzące do specyficznych neuronów wyróżnione są kolorami w dwóch warstwach.



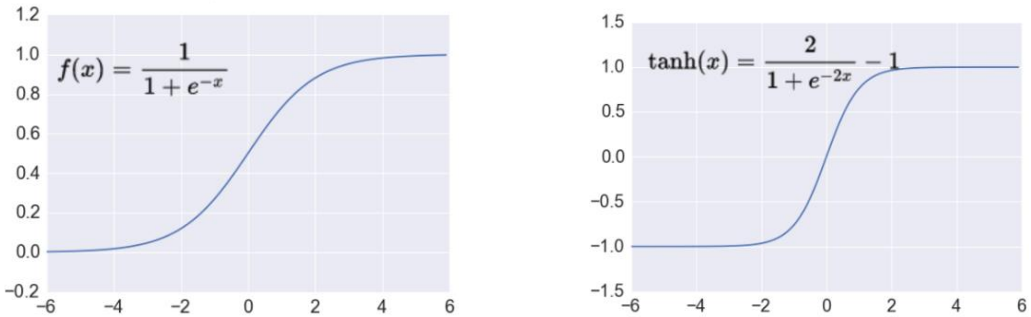
Aktywacja n-tego elementu w warstwie wyjściowej może być opisana w postaci matematycznej formuły:

$$a_n^L = \left[f\left(\sum_m w_{nm}^L \left[\ldots \left[f\left(\sum_i w_{ji}^1 x_i + b_j^1 \right) + b_k^2 \right] \ldots \right] + b_n^L \right) \right]$$

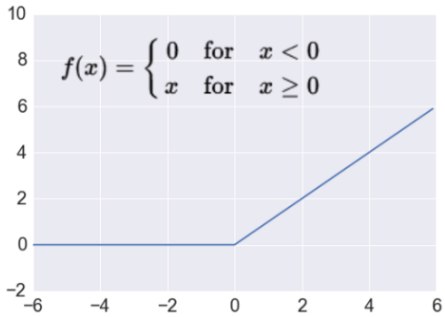
Sieć neuronowa realizuje taką funkcję matematyczną.
Jak pamiętamy z wykładu funkcja aktywacji musi być funkcją różniczkowalną. Mamy do wyboru wiele funkcji, z których najbardziej popularne pokazane są na rysunkach poniżej. Jedną z pierwszych była funkcja sigmoidalna Rys.4. Funkcja sigmoidalna ma dwie asymptoty i w przedziałach gdzie jest

prawie płaska gradient jest prawie równy 0. Podobnie wygląda kształt funkcji tangensa hiperbolicznego Rys.4b).

a) b)



Rys. 4. Klasyczne funkcje aktywacji a) sigmoidalna, b) tangensa hiperbolicznego (źródło: *http://adilmoujahid.com/images/activation.png*)



Rys.5 Funkcja aktywacji ReLU (źródło: *http://adilmoujahid.com/images/activation.png)*

Problem zanikającego gradientu powodował, że uczenie głębszych sieci było problematyczne. Dlatego w sieciach głębokich stosuje się najczęściej funkcję ReLU pokazaną na Rys. 5.

W warstwie wyjściowej, dla zadania klasyfikacji, które będziemy rozwiązywać w tym ćwiczeniu, informacja kodowana jest na zasadzie „1 z n". Oznacza to, że oczekujemy aby tylko 1 neuron miał wartość 1, ten który odpowiada rozpoznanej klasie, pozostałe neurony powinny mieć wartość równą 0. Aby umożliwić probabilistyczną interpretację wyniku otrzymanego na wyjściu, obecnie najczęściej stosuje się funkcję *softmax*.
Ogólnie możemy powiedzieć, że funkcja softmax bierze N wymiarowy wektor wartości rzeczywistych i produkuje inny N-wymiarowy wektor wartości rzeczywistych z przedziału (0,1), w taki sposób, że składowe wektora sumują się do 1.

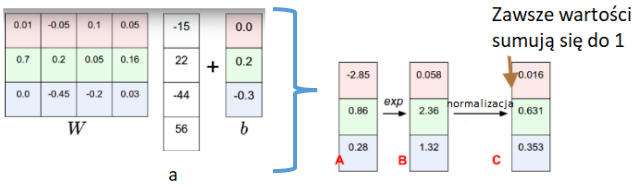
$$S(\mathbf{z}): \begin{bmatrix} z_1 \\ z_2 \\ \dots \\ z_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \dots \\ S_N \end{bmatrix}$$

Formuła określająca pojedynczy element wektora jest następująca:

$$S_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}$$

Łatwo zobaczyć (Rys. 6.), że wartości są zawsze dodatnie (z powodu eksponent) a w mianowniku mamy sumowanie po liczbach dodatnich, dlatego zakres jest ograniczony do przedziału (0,1). Na przykład, 3 elementowy wektor [1.0, 2.0, 3.0] jest transformowany do wektora [0,09, 0.24, 0.67] a więc względne relacje jeśli chodzi o wielkość składowych są zachowane a składowe sumują się do 1. Jeśli rozważany wektor zmienimy [1.0, 2.0, 5.0] otrzymamy wektor [0.02, 0.05, 0.93]. Zauważmy, że ostatni element wektora jest znacznie większy niż pierwsze dwa.

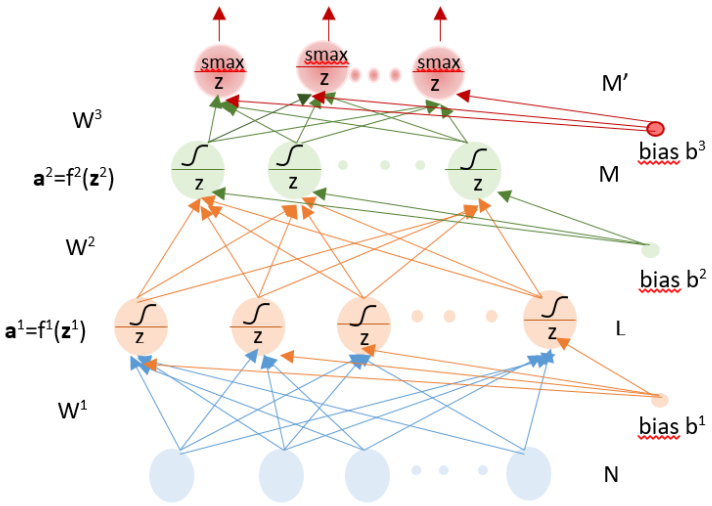
Intuicyjnie softmax jest miękką wersją funkcji maksimum, ale zamiast wybierać jedynie wartość maksymalną softmax wyraża składowe wektora w stosunku do całości.



Rys.6. Wizualizacja operacji wykonywanych w warstwie softmax (na podstawie slajdu z kursu CS 231n Stanford University)

Realizacja ćwiczenia

Na zajęciach należy zaimplementować prostą jednokierunkową sieć neuronową, do rozpoznawania cyfr ze zbioru MNIST oraz fazę przesłania wzorca do przodu (forward), tak aby otrzymać wyjście. Architekturę sieci przedstawia Rys. 7.



Rys.7. Architektura sieci, którą należy zaimplementować w ćwiczeniu 2.

W sieci wyróżniamy warstwę wejściową, dwie warstwy ukryte oraz warstwę wyjściową, która realizuje funkcję softmax.
Aplikacja powinna być napisana na tyle ogólnie, aby była możliwość:

b) użycia od 2-4 warstw,
c) użycia różnych funkcji aktywacji w warstwach ukrytych (sigmoidalna, tanh, ReLU),
d) użycia warstwy softmax (na Rys. 7. smax) w warstwie wyjściowej,
e) zmiany sposobu inicjalizowania wag (w tym ćwiczeniu przyjmujemy, że wagi będą inicjalizowane z rozkładu normalnego ze zmiennym odchyleniem standardowym),
f) Zmiany liczby neuronów w warstwach ukrytych,
g) przerwania uczenia i ponownego rozpoczęcia nauki od poprzednich wartości wag.

Dla sieci pokazanej na Rys. 7 **faza przesłania wzorca w przód** przedstawia się następująco.
Dla warstwy pierwszej:
Obliczenie całkowitego pobudzenia $\mathbf{z}^1 = \mathbf{W}^1\mathbf{x} \quad + \mathbf{b}^1$
Obliczenie aktywacji $\mathbf{a}^1 = f(\mathbf{z}^1)$
Dla warstwy drugiej:
Obliczenie całkowitego pobudzenia $\mathbf{z}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2$
Obliczenie aktywacji $\mathbf{a}^2 = f(\mathbf{z}^2)$
Dla warstwy trzeciej
Obliczenie całkowitego pobudzenia $\mathbf{z}^3 = \mathbf{W}^3\mathbf{a}^2 + \mathbf{b}^3$
Oblicz wyjście z sieci , poszczególne składowe wyjścia to :

$$\hat{y}_j = \frac{e^{z_j}}{\sum_{k=1}^{M\prime} e^{z_k}}$$

Za implementację architektury na zajęciach, można otrzymać 20% całkowitej liczby punktów.

Laboratorium 4: Implementacja metody uczenia dla sieci feedforward

Cel: rozszerzenie aplikacji zbudowanej na poprzednim laboratorium o możliwość uczenia sieci

Wprowadzenie teoretyczne

Jak wspomniano przy wprowadzeniu do poprzedniego laboratorium, jednokierunkowe sieci wielowarstwowe uczone są metodą nadzorowaną. Uczenie (trenowanie sieci) sprowadza się do znalezienia parametrów sieci θ, czyli wag i biasów, takich że $\forall i \in [1, N]: \mathrm{f}_\theta\ (\mathbf{x}_i) = \mathbf{y}_i$

Możemy to osiągnąć minimalizując błąd (koszt, funkcję straty) na zbiorze uczącym D, co możemy zapisać jako:

$$\min_\theta \mathrm{L}(f_\theta, D) = \min_\theta \frac{1}{P} \sum_{i=1}^{P} L(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

W tym wzorze L jest funkcją straty. Dla regresji używamy błędu średniokwadratowego:

$$L(f_\theta(\mathbf{x}_i), \mathbf{y}_i) = \frac{1}{2} \sum_{j=1}^{M} \left(f_{j,\theta}(\mathbf{x}) - \mathrm{y}_j\right)^2$$

Dla klasyfikacji do M klas funkcją straty jest ujemny logarytm szans (*negative log likelihood*)

$$L(f_\theta(\mathbf{x}_i), \mathbf{y}_i) = \sum_{j=1}^{M} -log\left(f_{j,\theta}(\mathbf{x})\right)\ \mathrm{y}_j$$

Najprostszą metodą uczenia jest metoda GD (Gradient Descent), kiedy uaktualnianie wag odbywa się po całej epoce uczenia.

Algorytm Gradient Descent (GD)

Zainicjuj losowo θ^0

Wykonaj

$$\theta^{t+1} = \theta^{t+1} - \gamma \frac{\partial L(f_\theta, D)}{\partial \theta}$$

dopóki

$$\left(\min_\theta \mathrm{L}(f_{\theta^{t+1}}, V) - \min_\theta \mathrm{L}(f_{\theta^t}, V)\right)^2 > \epsilon$$

gdzie V jest zbiorem walidacyjnym, γ jest współczynnikiem uczenia, L jest sumą gradientów, po całym zbiorze D.

W tym przypadku obliczenie gradientu jest kosztowne obliczeniowo, jeśli zbiór uczący jest duży. Problemem jest także odpowiedni dobór współczynnika uczenia, ale to będzie tematem ćwiczenia 3., w którym użyjemy bardziej zaawansowanych technik do optymalizacji współczynnika uczenia, takich jak rprop/rmsprop, adagrad czy adam.

Powiedzieliśmy, że GD jest kosztowne obliczeniowo, dlatego próbuje się obliczać gradient na części danych i taki algorytm nazywany jest SGD (od Stochastic Gradient Descent).

Algorytm Stochastic Gradient Descent SGD

Zainicjuj losowo θ^0

Wykonaj

Próbkuj przykłady$(\boldsymbol{x}', \mathbf{y}')$ ze zbioru D

$\theta^{t+1} = \theta^{t+1} - \gamma^t \frac{\partial L(f_\theta(\mathbf{x}'), \mathbf{y}')}{\partial \theta}$

dopóki

$\left(\min_\theta \mathrm{L}(f_{\theta^{t+1}}, V) - \min_\theta \mathrm{L}(f_{\theta^t}, V)\right)^2 > \epsilon$

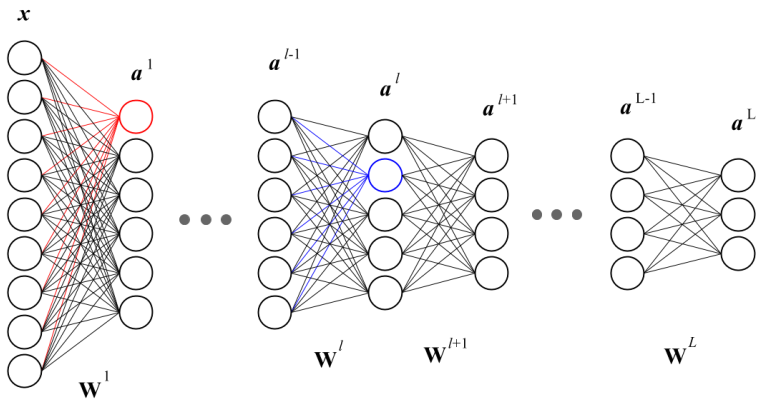
gdzie $\sum_{t=1} \gamma^t \to \infty$ oraz $\sum_{t=1} (\gamma^t)^2 < \infty$

SGD przyspiesza optymalizację dla dużych zbiorów, ale wprowadza szum do uaktualniania wag. W praktyce używa się tzw. minibatch - paczki wzorców. Paczka, w zależności od zbioru danych, liczy od kilku do kilkuset wzorców.

W obliczaniu gradientu stosujemy regułę łańcuchową. Przypomnijmy, że jeśli mamy do czynienia z funkcją złożoną *f*, która zależy of funkcji *g*, to pochodną obliczamy następująco:

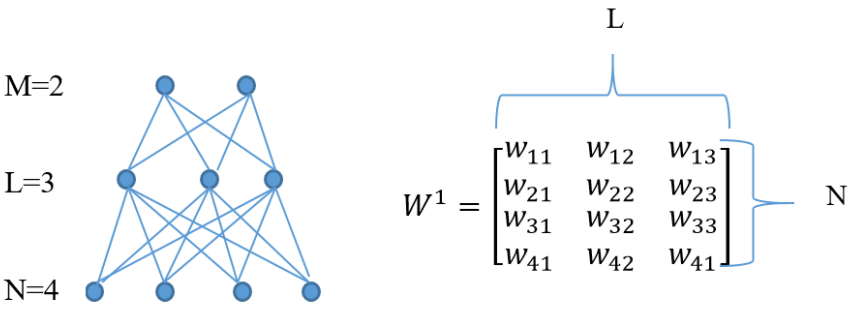
$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{g(x)}{\partial x}$$

Przypomnijmy ogólny schemat architektury sieci o liczbie warstw L. W tym przypadku macierz wag W^l odpowiada wagom na połączeniach między neuronami z warstwy l a neuronami warstwy l+1.



Rys. 8. Sieć wielowarstwowa zawierająca więcej warstw ukrytych

W macierzy tej wiersze odpowiadają neuronom w warstwie wcześniejszej l a kolumny neuronom w warstwie (l+1). Pokazuje to przykład na Rys. 9.



Rys. 9. Przykład zapisu wag w macierzy dla prostej architektury sieci

Jak pokazano na wykładzie będzie polegało na obliczeniu błędów w każdej warstwie począwszy od ostatniej do pierwszej, a po ich obliczeniu na adaptacji wag proporcjonalnie do błędu. Przyjmując L warstw w sieci i funkcję kosztu C (funkcja, którą będziemy minimalizować) błąd δ^L w ostatnie-tej warstwie sieci możemy zdefiniować począwszy od warstwy końcowej L jako:

$$\delta^L = \frac{\partial C}{\partial a^L} \odot f'(z^L)$$

gdzie $\mathbf{a}^L$ to wynik działania funkcji aktywacji w tej warstwie czyli $\mathbf{a}^L = f(\mathbf{z}^L)$ i ponieważ jest to ostatnia wartswa w sieci to jej wyjście $\hat{\mathbf{y}} = \mathbf{a}^L$. C Jest funkcją kosztu a znak $\odot$ odpowiada iloczynowi Hadamarda (mnożeniu odpowiadających sobie składowych)

Ogólnie dla l-tej warstwy błąd liczymy jako:

$$\boldsymbol{\delta}^l = (W^{l+1})^T \boldsymbol{\delta}^{l+1} \odot f'(\boldsymbol{z}^{l+1})$$

Możemy więc powiedzieć, ze dla dowolnej warstwy l ten błąd jest uzależniony od błędu warstwy wyższej, który pomnożony jest przez transponowaną macierz wag a następnie wykonywana jest operacja iloczynu Hadamarda przez pochodna funkcji aktywacji w tej warstwie. Obliczenie tych błędów stanowi podstawę do obliczania gradientu błędu po wagach w danej warstwie i dlatego należy tę operacje dobrze rozumieć. Załóżmy, że warstwa l ma K neuronów a warstwa l+1 ma M neuronów, wynik mnożenia pokazany jest poniżej. Zachęcam do zrobienia ćwiczenia na kartce papieru.

$$(\mathrm{W}^{l+1})^T \delta^{l+1} = \begin{bmatrix} w_{11}^{l+1} & \dots & w_{m1}^{l+1} & \dots & w_{M1}^{l+1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{1k}^{l+1} & \dots & w_{mk}^{l+1} & \dots & w_{Mk}^{l+1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ w_{1K}^{l+1} & \dots & w_{mK}^{l+1} & \dots & w_{MK}^{l+1} \end{bmatrix} \begin{bmatrix} \delta_1^{l+1} \\ \vdots \\ \delta_m^{l+1} \\ \vdots \\ \delta_M^{l+1} \end{bmatrix} = \begin{bmatrix} \sum_{m=1}^{M} \delta_m^{l+1} w_{m1}^{l+1} \\ \vdots \\ \sum_{m=1}^{M} \delta_m^{l+1} w_{mk}^{l+1} \\ \vdots \\ \sum_{m=1}^{M} \delta_m^{l+1} w_{mK}^{l-1} \end{bmatrix}$$

Schemat algorytmu uczenia, który poznaliśmy na wykładzie dla pojedynczego wzorca metoda GD wygląda następująco.

1. **Propagacja sygnału do przodu** – obliczamy pobudzenia każdego neuronu przez wykonywane iteracyjnie mnożenia macierzy wag i wektorów aktywacji z warstwy poprzedzającej a następnie docierających a następnie wykonujemy operacje na odpowiadających sobie składowych pobudzenia i funkcji aktywacji w każdej warstwie. Zapamiętujemy wynik.
2. **Obliczenie błędu w ostatniej warstwie L –** ta operacja wymaga obliczenia gradientu kosztu funkcji. Wyrażenie zależy od bieżącego wzorca (x i y) a także od wybranej funkcji kosztu (straty, błędu).
3. **Wykonaj propagację sygnału wstecz** – oblicz błędy dla neuronów w każdej warstwie. W tych obliczeniach także będą potrzebne pobudzenia neuronów (dlatego trzeba je było zapamiętać w trakcie propagacji sygnałów w przód). Tutaj również wykonywane są iteracyjne obliczenia macierzowo wektorowe.
4. **Oblicz pochodne funkcji kosztu** ze względu na wagi. W tym przypadku konieczne są również aktywacje każdego neuronu. W wyniku otrzymamy macierz o tych samych wymiarach jak macierz wag.
5. **Oblicz pochodne funkcji kosztu ze względu na bias.** Wynikiem będzie wektor kolumna.
6. **Zaktualizuj wagi** zgodnie z regułą Generalized Delta Rule (GDR)

Wyjaśnienia powyższe odnoszą się do zmiany wag sieci po przetworzeniu jednego wzorca. Jak użyć paczki wzorców? Teraz nasze wejście będzie reprezentowane w postaci macierzy, w której kolumny będą odpowiadać wzorcom, natomiast wiersze, to odpowiednie składowe wzorca. Przez wykonanie propagacji w przód. Pobudzenia i aktywacje będą również pamiętane w macierzach, gdzie indeks kolumny odpowiada indeksowi wzorca a indeks wiersza indeksowi neuronu. W macierzach będą tez pamiętane błędy dla różnych warstw I wzorce . Pochodne cząstkowe funkcji kosztu w odniesieniu do wag będą w trójwymiarowym tensorze o wymiarach [nr próbki, nr neuronu, nr wagi]

Ogólnie algorytm uaktualniania wag dla m wzorców w paczce możemy zapisać następująco:

- **Wejście x:** dla warstwy wejściowej ustawiamy aktywację $\mathbf{a}^1$ równą wzorcowi wejściowemu
- **Przesłanie wejścia w przód:** For each *l*=2,3,…,L oblicz

$$\mathbf{z}^l = \mathrm{W}^l\mathbf{a}^{l-1} + \mathbf{b}^l \text{ oraz } \mathbf{a}^l=\mathrm{f}(\mathbf{z}^l)$$

- **Błąd na wyjściu:** Obliczyć wektor

$$\boldsymbol{\delta}_{\mathbf{x}}^{\boldsymbol{L}} = \frac{\partial C_{\mathbf{x}}}{\partial a^L} \odot \boldsymbol{f}'(\mathbf{z}_{\mathbf{x}}^L)$$

- **Rzutowanie błędu wstecz**: For each l=L−1,L−2,…,2 oblicz

$$\boldsymbol{\delta}_{\mathbf{x}}^l = \left(\left(W^l\right)^T \boldsymbol{\delta}_{\mathbf{x}}^{l+1} \odot \boldsymbol{f}'(\mathbf{z}_{\mathbf{x}}^l) \right)$$

- **Uaktualnianie wag** For each l=L,L−1,…,2

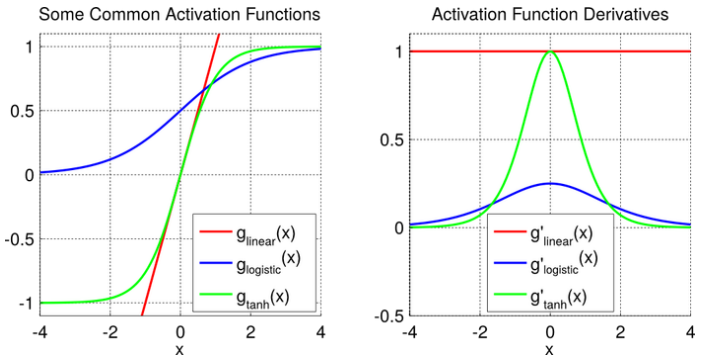
uaktualnianie wag i biasów zgodnie z regułą dla paczki wzorców

$$\mathrm{W}^l = \mathrm{W}^l - \frac{\alpha}{m} \sum_{\mathbf{x}} \boldsymbol{\delta}_{\mathbf{x}}^l \left(\mathbf{a}_{\mathbf{x}}^{l-1}\right)^T$$

$$\mathbf{b}^l = \mathbf{b}^l - \frac{\alpha}{m} \sum_{\mathbf{x}} \boldsymbol{\delta}_{\mathbf{x}}^l$$

m- jest liczbą wzorców w paczce, α współczynnikiem uczenia.

Jak widać, aby określić błąd w każdej warstwie musimy obliczyć pochodną funkcji aktywacji . Oznacza to, że używane funkcje muszą być różniczkowalne, To ograniczenie spełniają wszystkie podane w poprzednim laboratorium funkcje: sigmoidalna, tangensa hiperpobolicznego i ReLU, ale musimy znać ich pochodne.



Rys. 10. Funkcja liniowa g_{linear}, logistyczna $g_{logistic}$ i tangesa hiperbolicznego g_{tanh} oraz ich pochodne (źródło: https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-activation-functions/)

Na rysunku 10. Funkcje aktywacji oznaczane są przez g. Trzymając się jednak oznaczeń przyjętych w tej pracy, funkcje aktywacji będziemy dalej oznaczać jako f.

Funkcja liniowa: Funkcję liniową dla pobudzenia *z* możemy zapisać jako

$$f_{linear}(z) = z$$

a jej pochodna jest równa 1, dla przypadku jednowymiarowego. W przypadku większej liczby wymiarów, gradient jest wektorem współrzędnych o wartości 1.
Funkcja sigmoidalna (funkcja logistyczna): Funkcja logistyczna $f_{logistic}$ wyraża się wzorem:

$$f_{logistic} = \frac{1}{1 + e^{-z}}$$

gdzie z jest pobudzeniem neuronu w tym przypadku.
Pochodna funkcji logistycznej jest równa

$$f'_{logistic} = f_{logistic}(1 - f_{logistic})$$

Warto jest więc pamiętać w trakcie przetwarzania w przód wartości funkcji aktywacji dla danej warstwy, dzięki czemu unikamy w ten sposób ponownego obliczania wartości funkcji aktywacji do zmiany wag, która wymagałaby kosztownego obliczania funkcji eksponencjalnej (logistycznej).

Funkcja tangensa hiperbolicznego: Funkcja tangensa hiperbolicznego ma podobny kształt do funkcji logistycznej ale wartości tej funkcji są w zakresie (-1,1) i wyraża się wzorem:

$$f_{tanh} = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

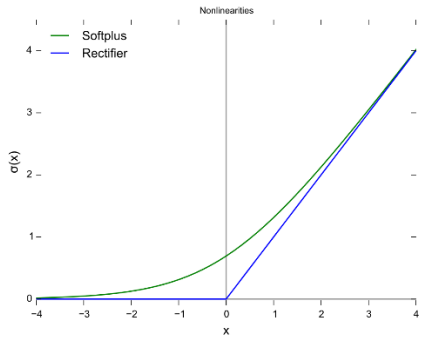
Warto też zauważyć, że jedynie wartości bliskie 0 są mapowane na wartości wyjściowe bliskie 0.
Pochodna funkcji:

$$f'_{tanh} = (1 - (f_{tanh})^2)$$

A więc i w tym przypadku, dobrze jest przechowywać wartości funkcji aktywacji obliczone w fazie przesłania w przód w pamięci, aby uniknąć powtarzania kosztownych obliczeń w fazie rzutowania błędów wstecz.

Funkcja ReLU: W sieciach głębokich najczęściej wykorzystuje się jako funkcje aktywacji funkcję ReLU:

$$f_{ReLU}(z) = \max(0, z)$$



Rys. 11. Funkcja ReLU i softplus (źródło: https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#/media/File:Rectifier_and_softplus_functions.svg)

Jak możemy zobaczyć na Rys. 11, funkcja ta jest nieróżniczkowalna w 0. Dla wartości większych od 0 pochodna jest równa 1 dla mniejszych od 0, pochodna jest równa 0.

Istnieją dwa sposoby rozwiązania tego problemu. Pierwszy, to przypisanie arbitralnej wartości dla ReLU(0), np. 0 lub 0,5. Drugi to użycie funkcji softplus, która w pewnym stopniu aproksymuje ReLU i jest w całym zakresie wartości różniczkowalna (Rys. 11). Wyraża się ona następującym wzorem:

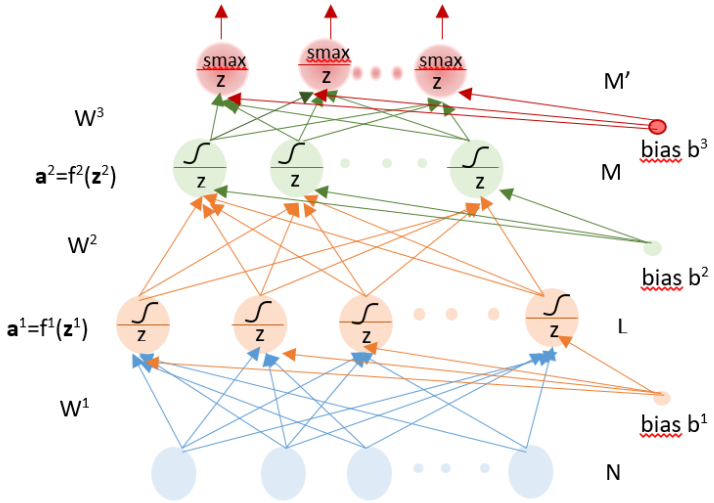
$$f'_{softplus}(z) = \ln(1 + e^z)$$

A jej pochodną jest to funkcja logistyczna

$$f'_{softplus}(z) = \frac{1}{1 + e^{-z}}$$

Realizacja ćwiczenia

Przypomnijmy oznaczenia i architekturę zbudowanej sieci jednokierunkowej



Na poprzednim laboratorium zaimplementowana została sieć, której architektura pokazana na powyższym rysunku. Możliwe jest też przesłanie wzorca przez wszystkie warstwy, tak aby policzyć wyjście $\hat{\mathbf{y}}$ (z sieci (faza przesłania w przód).

Na tym laboratorium zadaniem jest implementacja możliwości uczenia sieci, czyli znajdowania jej parametrów θ (wag i biasów). Ich poszukiwanie odbywa się poprzez minimalizację funkcji straty L (inaczej nazywanej kosztem lub błędem).

Przyjmiemy, że w tym ćwiczeniu funkcją straty jest ujemny logarytm szansy (ang. negative log likelyhood), czyli:

$$L(f_\theta(\mathbf{x}_i), \mathbf{y}_i) = \sum_{j=1}^{M} -log\hat{y_j}\ \ y_j$$

Najpierw musimy policzyć stratę L w warstwie wyjściowej a następnie rzutować błędy wstecz.

W pierwszej kolejności obliczamy gradient funkcji straty dla warstwy trzeciej.

$$\frac{\partial L(f(\mathbf{z}^3), \mathbf{y})}{\partial W^3} = \frac{\partial L\big((f(\mathbf{z}^3), \mathbf{y})\big)}{\partial\ \mathbf{z}^3}\ \frac{\partial\ \mathbf{z}^3}{\partial W^3}$$

Gradient z funkcji softmax jest równy

$$\frac{\partial L\big((f(\mathbf{z}^3), \mathbf{y})\big)}{\partial\ \mathbf{z}^3} = -(\mathbf{y} - f(\mathbf{z}^3)) =$$

Gdzie $\mathbf{y}$ jest kodowane jako „1 z n”.

Przypomnijmy, że $\mathbf{z}^3 = \mathbf{W}^3\mathbf{a}^2 + \mathbf{b}^3$, więc gradient $\boldsymbol{z}^3$ względem ∂W^3wyraża się następująco:

$$\frac{\partial\ \mathbf{z}^3}{\partial W^3} = \big(\mathbf{a^2}\big)^T$$

Po połączeniu obu wyrażeń otrzymujemy

$$\frac{\partial L(f(\mathbf{z}^3), \mathbf{y})}{\partial W^3} = -(\mathbf{y} - f(\mathbf{z}^3))\big(\mathbf{a^2}\big)^T$$

Teraz przechodzimy do warstwy drugiej

$$\frac{\partial L(f(\mathbf{z}^3), \mathbf{y})}{\partial \mathbf{a}^2} = \frac{\partial L\big((f(\mathbf{z}^3), \mathbf{y})\big)}{\partial\ \mathbf{z}^3}\ \frac{\partial\ \mathbf{z}^3}{\partial \mathbf{a}^2}$$

Biorąc pod uwagę, że $\mathbf{z}^3 = \mathbf{W}^3\mathbf{a}^2 + \mathbf{b}^3$ ostatni element będzie macierzą wag$\mathbf{W}^3$

A więc równanie możemy zapisać jako:

$$\frac{\partial L(f(\mathbf{z}^3),\mathbf{y})}{\partial \mathbf{a}^2} = (\mathbf{W}^3)^T \frac{\partial L\big((f(\mathbf{z}^3),\mathbf{y})\big)}{\partial\, \mathbf{z}^3}$$

Gradient po wagach w warstwie drugiej

$$\frac{\partial L(f(\mathbf{z}^3),\mathbf{y})}{\partial W^2} = \frac{\partial L(f(\mathbf{z}^3),\mathbf{y})}{\partial \mathbf{a}^2} \frac{\partial\, \boldsymbol{a}^2}{\partial \boldsymbol{z}^2} \frac{\partial\, \boldsymbol{z}^2}{\partial W^2}$$

Pochodna funkcji aktywacji
Trzeba wstawić odpowiednie wartości w zależności od zaimplementowanej funkcji aktywacji

Wiemy, że $\mathbf{z}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2$, więc ostatni element gradientu to będzie $(\mathbf{a}^1)^T$.
Gradient po wagach w warstwie pierwszej

Należy pamiętać o użyciu właściwego wzoru na pochodną w stosunku do zaimplementowanej funkcji aktywacji.
Dalej powtarzamy ten sam schemat dla warstwy pierwszej

Faza przesłania w tył jest powtarzalną operacją stosowania reguły łańcuchowej
Dlatego jest potencjalnym miejscem do istnienia wielu błędów w aplikacji. Sprawdzenie gradientu jest jedną z możliwości ich znalezienia. Poniżej przedstawiony jest przydatny do tego celu algorytm Gradient checking.

Algorytm Gradient Checking

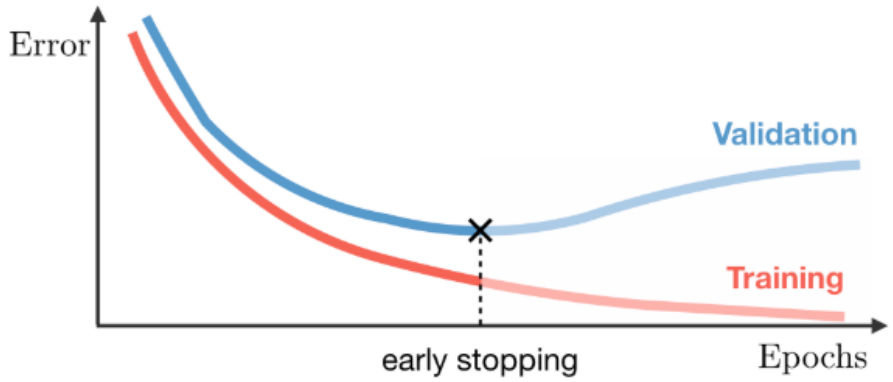
Zainicjuj losowo wartości parametrów $\theta = (W^1, b^1, W^2, \dots)$
Losowo zainicjuj $\mathbf{x}$ oraz $\mathbf{y}$
Oblicz analitycznie gradient używając propagacji wstecznej $g_{analytic} = \frac{\partial L(f_\theta(\mathbf{x}),\mathbf{y})}{\partial \theta}$
For i in # θ
$\hat{\theta} = \theta$
$\hat{\theta} = \hat{\theta} + \varepsilon$
$g_{numeric} = \frac{L(f_{\hat{\theta}}(\mathbf{x}),\mathbf{y}) - L(f_\theta(\mathbf{x}),\mathbf{y})}{\varepsilon}$
Musi być spełnione: $(\|g_{numeric} - g_{analytic}\| < \epsilon$

Laboratorium 5 - Wykonanie badań eksperymentalnych

Cel: Ocena skuteczności rozpoznawania cyfr ze zbioru MNIST przez zbudowany model przy różnych wartościach hiperparametrów

Wprowadzenie teoretyczne

Istnieją różne miary oceny klasyfikatorów. Ocena zawsze przeprowadzana jest z użyciem zbioru testowego, gdyż nowe wzorce zazwyczaj nie będą takie same jak dane uczące. Zazwyczaj dostępny zbiór danych dzieli się na zbiór uczący, walidujący i testowy. Zbiór walidujący służy do sprawdzenia czy model nie dopasował się zbytnio do specyfiki danych uczących.
Najczęściej sieć uczona jest w trybie wczesnego zatrzymywania (ang. *early stopping*). Ta technika polega na tym, że w trakcie uczenia obserwujemy błąd sieci, co epokę obliczamy także błąd na zbiorze walidującym. Jeśli błąd na zbiorze walidującym zaczyna rosnąć o wartość wyższą niż dopuszczalna, przerywamy uczenie i wracamy do wag, przy których błąd na zbiorze walidującym był najmniejszy.



Rys. 12. Wczesne zatrzymywanie uczenia (ang. early stopping) (źródło: Stanford kurs CS230)

Jest to przedstawione na Rys. 12., który pokazuje epokę, gdzie należy zakończyć uczenie sieci, mimo, że błąd na zbiorze uczącym ciągle maleje.
W zależności od postawionego celu będziemy używać różnych miar w celu oceny klasyfikatora. Może to być: trafność klasyfikacji (predictive accuracy), szybkość i skalowalność, czas uczenia.
Trafność klasyfikacji możemy zdefiniować jako:

$$accuracy = \frac{N_c}{N_t}$$

gdzie: N_c to liczba poprawnie zaklasyfikowanych wzorców ze zbioru testowego,
N_t to liczba wszystkich wzorców w zbiorze testowym.

Błąd klasyfikacji definiuje się następująco:

$$er_c = \frac{N_t - N_c}{N_t}$$

Inne możliwości analizy to macierz pomyłek (ang. confusion matrix), koszty pomyłek. Macierz pomyłek przy klasyfikacji do różnych klas pokazuje w formie macierzy r x r ile przykładów do której klasy zostało zaklasyfikowanych. Precyzyjniej mówiąc, wiersze macierzy odpowiadają klasom wynikającym ze zbioru testowego a kolumny klasom przewidywanym przez klasyfikator. Wartość n_{ij} na przecięciu wiersza *i* oraz kolumny j określa liczbę wzorców należących oryginalnie do klasy i-tej, a zaliczonej do klasy j-tej. W tabeli poniżej wyszczególniono jedynie dwie przykładowe wartości n_{23}

i n_{45}. Pierwsza mówi ile przykładów z klasy drugiej zostało zaklasyfikowanych przez sieć do klasy trzeciej, druga ile przykładów z klasy czwartej sieć zaklasyfikowała do klasy piątej.

	Przewidywane klasy					
Oryginalne klasy	K_1	K_2	K_3	K_4	K_5	K_6
K_1	…			…		…
K_2			n_{23}			
K_3	…					
K_4					n_{45}	
K_5	…		…			…

Dla klasyfikacji binarnej (wtedy, gdy mamy tylko dwie klasy) często używa się miary *sensitivity* (specyficzność) i *specificity* (wrażliwość/czułość) i krzywej ROC. W tym przypadku wartości w poszczególnych komórkach tabeli oznaczane są jako TP (true positive), FN (fase negative), FP(false positive), TN (true negative).

Oryginalne klasy	Przewidywane klasy decyzyjne	
	Pozytywna	Negatywna
Pozytywna	*TP*	*FN*
Negatywna	*FP*	*TN*

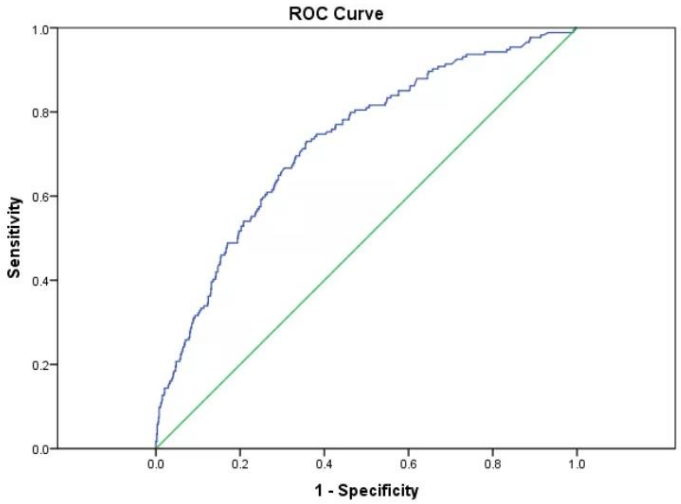
Czułość definiujemy jako:

$$sensitivity = \frac{TP}{TP + FN}$$

natomiast specyficzność jako

$$specificity = \frac{TN}{FP + TN}$$

Krzywa ROC (Receiver Operating Characteristic) jest bardzo użytecznym narzędziem do oceny i porównywania modeli. Krzywa ta pokazuje sensitivity w zależności od (1-specificity)



Rys. 13. Krzywa ROC

Najlepiej wybrać takie hiperparametry, przy których jest wysoka miara *sensitivity* i niska wartość (1-*specificity*). Wtedy największa liczba wzorców z oryginalnej klasy będzie rozpoznana właściwie i tylko niewielka liczba z klasy przeciwnej będzie rozpoznana jako klasa pozytywna (oryginalna).

Warto również wspomnieć o roli wielkości paczki wzorców w algorytmie SGD. Wybór odpowiedniego rozmiaru paczki wzorców wpływa na zbieżność funkcji kosztu, znalezienia parametrów sieci i możliwości uogólniania. W praktyce określamy tę wartość w procesie przeszukiwania wartości hiperparametru, pamiętając o następujących zasadach:

- Wielkość paczki określa częstość uaktualnień wag. Im mniejszy paczka wzorców, tym częściej będą aktualizowane wagi
- Im większa paczka wzorców, tym bardziej dokładny jest gradient funkcji kosztu, to znaczy jest bardziej prawdopodobne, że kierunek aktualizacji wag będzie skierowany do lokalnej pochyłości funkcji kosztu
- Większe paczki (ale nie za duże) mogą poprawiać zrównoleglenie uczenia na GPU

Realizacja ćwiczenia

W zasadzie poprzednie dwa laboratoria pozwoliły implementacje sieci wraz z metodą uczenia. Należy pamiętać, aby w zbudowanej aplikacji:

- istniała możliwość budowy sieci o dowolnej liczbie neuronów w warstwie ukrytej,
- istniała możliwość rozpoczęcia nauki od losowych wag z różnych zakresów,
- istniała możliwość przerwania uczenia i ponownego rozpoczęcia nauki od poprzednich wartości wag (early stopping),
- istniał wybór różnych funkcji aktywacji (tanh lub ReLU) i możliwość obserwację bieżącego błędu uczenia sieci.

Dobrze też byłoby zapewnić serializację badań.

W celu realizacji ćwiczenia należy wykonać eksperymenty, w których należy zbadać:
a. szybkość uczenia (w epokach) i skuteczność w przypadku różnej liczby neuronów w warstwie ukrytej,
b. wpływ różnych wartości współczynniki uczenia,
c. wpływ wielkości paczki (batcha),
d. wpływ inicjalizacji wartości wag początkowych
e. wpływ funkcji aktywacji (tanh lub ReLU)

Sposób oceny:

Implementacja modelu architektury na zajęciach - 20% całkowitej liczby punktów przypisanej do zadania.
Implementacja metody uczenia na zajęciach - 20% całkowitej liczby punktów przypisanej do zadania.
Przebadanie szybkości uczenia (w epokach) i skuteczności w przypadku różnej liczby neuronów w warstwie ukrytej – 10% całkowitej liczby punktów przypisanej do zadania.
Przebadanie wpływu różnych wartości współczynniki uczenia 10% całkowitej liczby punktów przypisanej do zadania
Przebadanie wpływu wielkości paczki (batcha) – 10% całkowitej liczby punktów przypisanej do zadania
Przebadanie wpływu inicjalizacji wartości wag początkowych -10% całkowitej liczby punktów przypisanej do zadania
Przebadanie wpływu funkcji aktywacji (tanh lub ReLU) -20% całkowitej liczby punktów przypisanej do zadania.

Bonus za implementację i sprawdzenie poprawności gradientu (algorytm gradient checking)

Przesłanie raportu do prowadzącego do dnia ……

Interesujące linki:

https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d
https://peterroelants.github.io/posts/cross-entropy-softmax/
http://bigstuffgoingon.com/blog/posts/softmax-loss-gradient/
https://web.stanford.edu/class/cs224n/readings/gradient-notes.pdf