SIECI NEURONOWE – laboratorium

Ćwiczenie 2 – sieć wielowarstwowa uczona metodą propagacji wstecznej

60 pkt

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

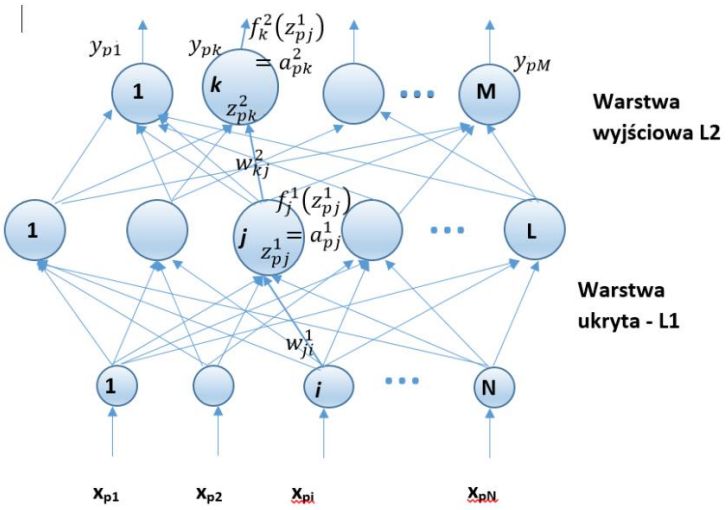
UWAGA: w tym ćwiczeniu można używać pakietu do mnożenia macierzy (numpy) jednak zbudowanie architektury sieci jak również implementacja metody uczenia musi być wykonana samodzielnie.

Laboratorium 3

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:

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

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

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

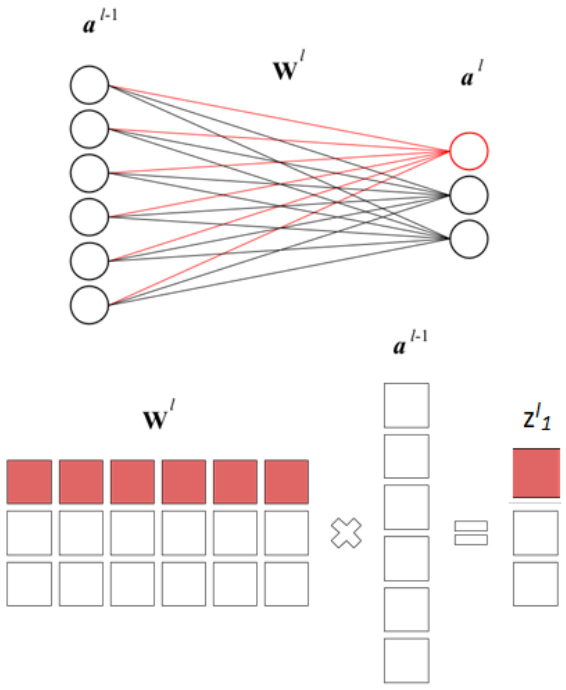
Jednak znacznie lepiej (i szybciej jeśli chodzi o obliczenia) jest używać notacji wektorowej.

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:

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

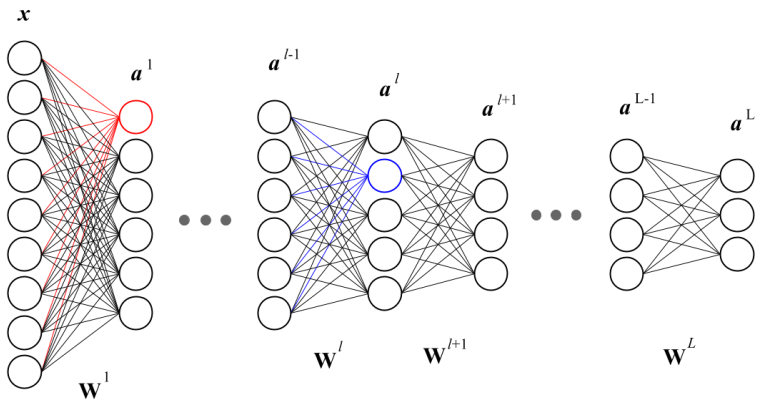
Powyższe mnożenie macierzy możemy zwizualizować jak na Rys.2., gdzie wprowadzony jest nowy wektor $\mathbf{z}^l$ oznaczający całkowite pobudzenie neuronu a następnie do tego wektora stosujemy funkcję aktywacji f, wykonując przekształcenie przez tę funkcję każdej składowej wektora.

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



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.

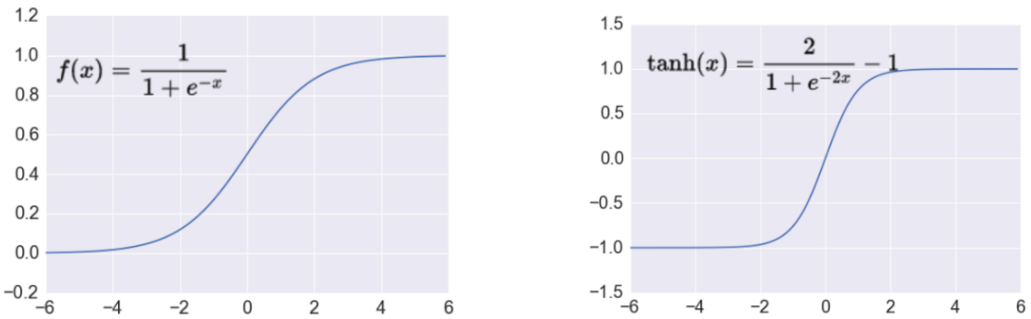


Aktywacja n-tego elementu w warstwie wyjściowej może być opisana w postaci matematycznej formuły:

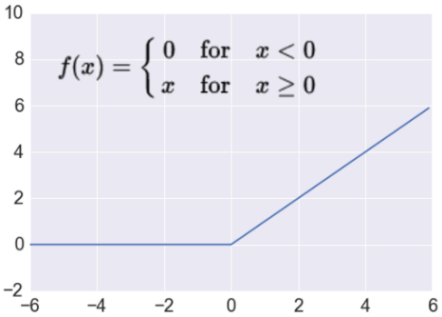
$$a_n^L = \left[f\left(\sum_m w_{nm}^L \left[\ldots \left[f\left(\sum_i w_{ji}^1 x_i + b_j^1 \right) + b_k^2 \right] \ldots \right] + b_n^L \right) \right]$$

Sieć neuronowa realizuje 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.

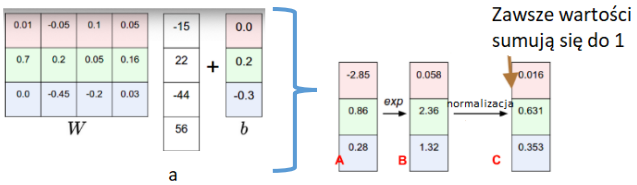
$$S(\mathbf{z}): \begin{bmatrix} z_1 \\ z_2 \\ \dots \\ z_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \dots \\ S_N \end{bmatrix}$$

Formuła określająca pojedynczy element wektora jest następująca:

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

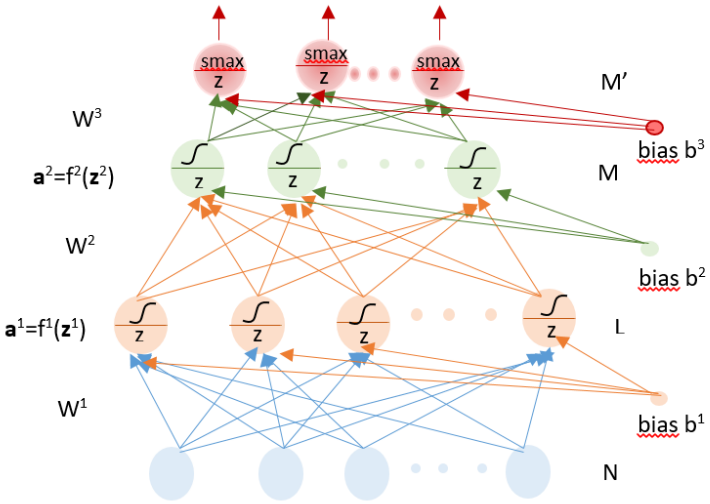
Łatwo zobaczyć, że wartości są zawsze dodatnie (z powodu eksponent) a w mianowniku mamy sumowanie po liczbach dodatnich, dlatego zakres jest ograniczony do przedziału (0,1). Na przykład, 3 elementowy wektor [1.0, 2.0, 3.0] jest transformowany do wektora [0,09, 0.24, 0.67] a więc względne relacje jeśli 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. 6.



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. 6. 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 $\mathbf{z}^1 = \mathbf{W}^1\mathbf{x} + \mathbf{b}^1$
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 :

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

Za realizację zadania 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 każdym wzorcu.

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.
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

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

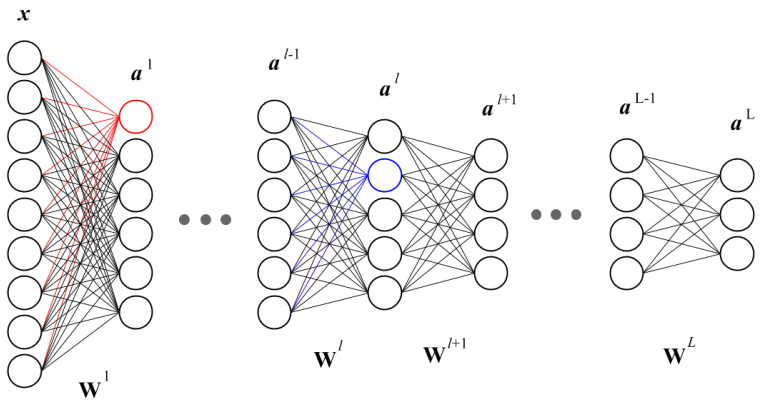
gdzie $\sum_{t=1}\gamma^t \to \infty$ oraz $\sum_{t=1}(\gamma^t)^2 < \infty$

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.

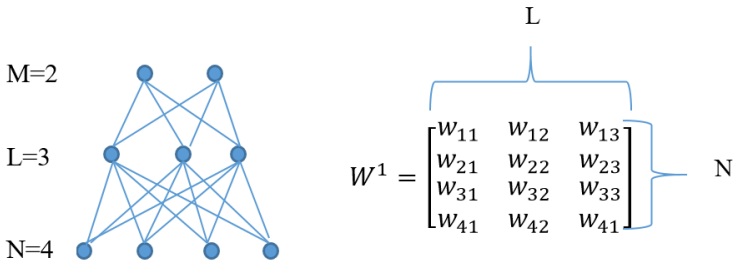
W obliczaniu gradientu stosujemy regułę łańcuchową. Przypomnijmy, że jeśli mamy do czynienia z funkcją złożoną *f*, która zależy of funkcji *g*, to pochodną obliczamy następująco:

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

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.



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



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.

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

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 l 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
  $\mathbf{z}^l = \mathrm{W}^l\mathbf{a}^{l-1} + \mathbf{b}^l$ 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

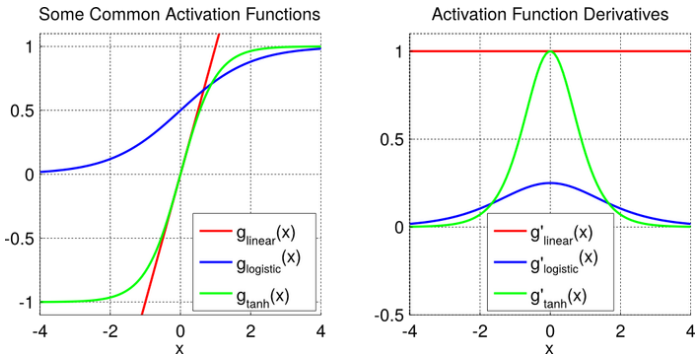
uaktualnianie wag i biasów zgodnie z regułą dla paczki wzorców

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

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

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.



https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-activation-functions/

Funkcja liniowa: $g_{\text{linear}}(z) = z$ The derivative of g_{linear} , g'_{linear}, is simply 1, in the case of 1D inputs. For vector inputs of length D the gradient is $\vec{1}^{1xD}$, a vector of ones of length D.

Funkcja sigmoidalna (logistic function) $g_{\text{logistic}}(z) = \frac{1}{1+e^{-z}}$

Here we see that $g'_{logistic}(z)$ evaluated at z is simply $g_{logistic}(z)$ weighted by 1-minus $g_{logistic}(z)$. This turns out to be a convenient form for efficiently calculating gradients used in neural networks: if one keeps in memory the feed-forward activations of the logistic function for a given layer, the gradients for that layer can be evaluated using simple multiplication and subtraction rather than performing any re-evaluating the sigmoid function, which requires extra exponentiation.

Funkcja tangensa hiperbolicznego .Like the log$(-1, 1)$moid, the tanh function is also sigmoidal ("s"-shaped), but instead o ts values that range . Thus strongly negative inputs to the tanh will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs. These properties make the network less likely to get "stuck" during training. Calculating the gradient for the tanh function also uses the quotient rule:

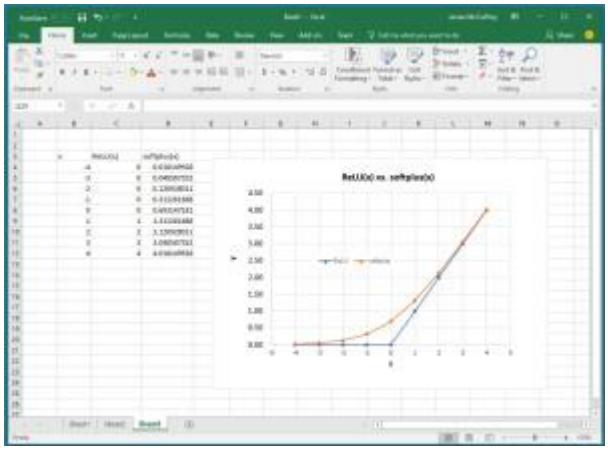
Similar to the derivative for the logistic sigmoid, the derivative of $g_{\tanh}(z)$ is a function of feed-forward activation evaluated at z, namely $(1 - g_{\tanh}(z)^2)$. Thus the same caching trick can be used for layers that implement tanh activation functions.

A NN uses one or more internal activation functions. One common activation function is the logistic sigmoid, logsig(x) = 1.0 / (1.0 + e^-x). Back-propagation requires the Calculus derivative of the activation function. If y = logsig(x), then the Calculus derivative is y' = e^-x / (1.0 + e^-x)^2 and by a very cool, non-obvious algebra coincidence y' = y * (1 – y).

But for deep neural networks, a common activation function is ReLU(x) = max(0, x). If you graph y = ReLU(x) you can see that the function is mostly differentiable. If x is greater than 0 the derivative is 1 and if x is less than zero the derivative is 0. But when x = 0, the derivative does not exist.

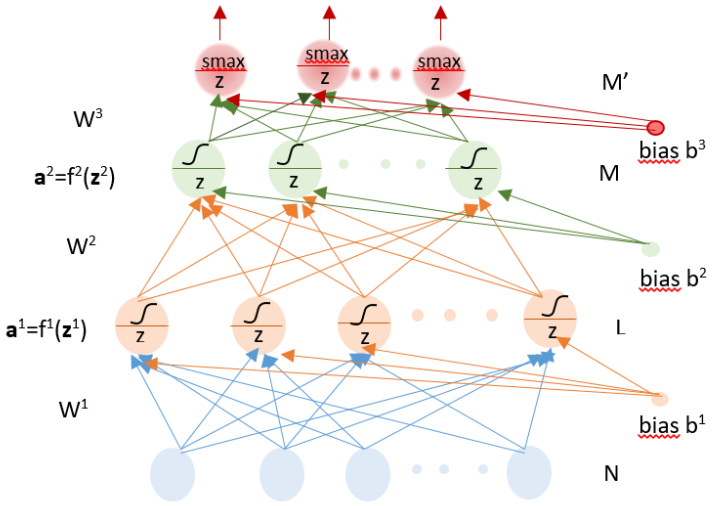
There are two ways to deal with this. First, you can just arbitrarily assign a value for the derivative of y = ReLU(x) when x = 0. Common arbitrary values are 0, 0.5, and 1. Easy!

A second alternative is, instead of using the actual y = ReLU(x) function, use an approximation to ReLU which is differentiable for all values of x. One such approximation is called softplus which is defined y = ln(1.0 + e^x) which has derivative of y' = 1.0 / (1.0 + e^-x) which is, remarkably, the logistic sigmoid function. Neat!



When I implement a deep NN from scratch, I usually use the arbitrary-value-when-x-equals-zero approach. I have never seen any research that looks at which of the two ways to deal with y = ReLU(x) being non-differentiable at 0, is better.

Realizacja ćwiczenia:
Przypomnijmy oznaczenia i architekturę zbudowanej sieci jednokierunkowej



Na poprzednim laboratorium zaimplementowana została sieć, której architektura pokazana jest na Rys.8 . 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.

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

Gradient z funkcji softmax jest równy

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

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:

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

Po połączeniu obu wyrażeń otrzymujemy

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

Teraz przechodzimy do warstwy drugiej

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

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

WYPISAC POCHODNE FUNKCJI AKTYWWACJI
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 **x** oraz **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: $(\|g_{numeric} - g_{analytic}\| < \epsilon$

optimization via gradient descent (gd)
- optimization via stochastic gradient descent (sgd)
- gradient checking code (!!!)
- weight initialization with random noise (!!!) (use normal distribution with changing std. deviation for now)
- Bonus points for testing some advanced

Napisać o wielkości batcha!!!!!!!!!!!!!!

Batch size

Batch size is the number of data points used to train a model in each iteration. Typical small batches are 32, 64, 128, 256, 512, while large batches can be thousands of examples.

Choosing the right batch size is important to ensure convergence of the cost function and parameter values, and to the *generalization* of your model. Some research has considered how to make the choice, but there is no consensus. In practice, you can use a *hyperparameter search*.

Research into batch size has revealed the following principles:

- Batch size determines the frequency of updates. The smaller the batches, the more, and the quicker, the updates.
- The larger the batch size, the more accurate the gradient of the cost will be with respect to the parameters. That is, the direction of the update is most likely going down the local slope of the cost landscape.
- Having larger batch sizes, but not so large that they no longer fit in GPU memory, tends to improve parallelization efficiency and can accelerate training.
- Some authors (Keskar et al., 2016) have also suggested that large batch sizes can hurt the model's ability to generalize, perhaps by causing the algorithm to find poorer local optima/plateau.

In choosing batch size, there's a balance to be struck depending on the available computational hardware and the task you're trying to achieve.

Early stopping

According to Geoff Hinton: "*Early stopping (is) beautiful free lunch*" (NIPS 2015 Tutorial slides, slide 63). You should thus always monitor error on a validation set during training and stop (with some patience) if your validation error does not improve enough.

o dowolnej liczbie neuronów w warstwach,
umożliwiać rozpoczęcie nauki od losowych wag z różnych zakresów,
przerwanie uczenia i ponowne rozpoczęcie nauki od poprzednich wartości wag,
powinien umożliwiać korekcję wag z momentum i bez,
z różnymi wersjami funkcji kosztu (kryterium optymalizacji) w metodzie propagacji wstecznej oraz różnymi funkcjami aktywacji itd.
Dobrze też byłoby zapewnić serializację badań i obserwację bieżącego błędu uczenia sieci.

Proszę pamiętać, że sieć używamy w trzech trybach pracy:
Uczenia,
Testowania, walidacji
Tryb pracy dla użytkownika kiedy może on podawać dowolne wzorce i sprawdzać odpowiedź sieci
I takie tryby pracy powinna umożliwiać Wasza aplikacja.

Next step

This explanation is how to train a network using only one training sample at a time, but how to do it using batch learning? What we can do is just putting the inputs of the training samples as *columns in a matrix*, doing the forward propagation the *input sums* and *activations* will also be in matrices, where *column index is the sample index, and the row index is the neuron index*. Similarly the error signals for different layers and samples will be in matrices. However, the derivatives of the cost function with respect to the weights will be in a three-dimensional tensor using the dimensions `[sample_idx, neuron_idx, weight_idx]`. Reducing a sum over the samples in this tensor will give the gradient matrix for the weights in the actual layer, and the weights can be updated using the delta rule.

Należy przebadać:
a. szybkość uczenia w przypadku liczby neuronów w warstwie ukrytej,
b. zastosować różne współczynniki uczenia,
c. przetestować uczenie z momentum i bez,
d wpływ liczebności zbioru uczącego
e wpływ inicjalizacji wartości wag początkowych

Przebadanie wpływu rozmiaru warstwy
Przebadanie wpływu wielkości paczki (batcha)
Przebadanie wpływu ReLU

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

Given the true label $Y = y$, the only non-zero element of the 1-hot vector $p(x)$ is at the y index, which in practice makes the $p(x)$ vector a selector for the y index in the $q(x)$ vector. Therefore, the loss function for a single sample then becomes:

$$Loss = -\log(q_y) = -\log(\frac{e^{z_y}}{\sum_j e^{z_j}}) = -z_y + \log\sum_j e^{z_j}$$

Calculating the derivative for each z_i:

$$\begin{aligned}
\nabla_{z_i} Loss &= \nabla_{z_i}(-z_y + \log\sum_j e^{z_j}) \\
&= \nabla_{z_i} \log\sum_j e^{z_j} - \nabla_{z_i} z_y \\
&= \frac{1}{\sum_j e^{z_j}} \nabla_{z_i} \sum_j e^{z_j} - \nabla_{z_i} z_y \qquad \text{from} \quad \frac{d}{dx} ln[f(x)] = \frac{1}{f(x)} \frac{d}{dx} f(x) \\
&= \frac{e^{z_i}}{\sum_j e^{z_j}} - \nabla_{z_i} z_y \\
&= q_i - \nabla_{z_i} z_y \\
&= q_i - \mathbb{1}(y = i) \\
&\text{where} \quad \mathbb{1}(y = i) = \begin{cases} 1 & \text{if y=i} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

These results show:

- $\nabla_{z_y} Loss = q_y - 1$
  The gradient for the true label's logit is non-positive and decrease proportionally in magnitude as q_y increases.
- $\nabla_{z_i} Loss = q_i \quad \forall i \neq y$