

Historia — pierwsze modele sztucznych sieci neuronowych

Historycznie pierwszy model neuronu wprowadzony w 1943 (McCulloch i Pitts) potrafił rozpoznawać dwie kategorie obiektów na podstawie progowania wartości funkcji $f(\mathbf{x}) = \sum_i w_i x_i$. Jednak wagi musiały być wybrane przez operatora.

Pod koniec lat 50-tych XX-ego wieku pojawił się perceptron (Rosenblatt), który potrafił uczyć się poprawnych wag na podstawie próbek różnych kategorii (ale tylko dla sieci jednowarstwowej, wejściowo-wyjściowej).

Prawie równocześnie wprowadzono (Widrow i Hoff) model neuronu ADALINE (*Adaptive Linear Neuron*), który zwracał wartość $f(\mathbf{x})$ próbując przewidzieć liczbę rzeczywistą na podstawie wartości wejść, i sam uczył się swoich wag z danych.

Historia ANN: pierwsza fala entuzjazmu

Te wczesne modele zapoczątkowały pierwszą rewolucyjną falę zainteresowania sztucznymi sieciami neuronowymi. Ta rewolucja pojawiła się w czasach wczesnych komputerów i bazowała głównie na obietnicach i przewidywaniach, a nie na konkretnych projektach i zastosowaniach.

Cytat z New York Times, 8 lipca, 1958, “New Navy Device Learns By Doing”

“The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence ... Dr. Frank Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers”

Historia ANN: pierwsze rozczarowanie

Nietrudno zrozumieć, że szybko nastąpiło wygaszenie tego entuzjazmu, zwłaszcza po opublikowaniu słynnego artykułu Marvina Minsky'ego i Seymoura Paperta "*Perceptrons*," zawierającego techniczną i krytyczną analizę możliwości sztucznych sieci neuronowych. Jednym z kluczowych argumentów była niezdolność sieci neuronowej (jednowarstwowej) do obliczenia tak prostej funkcji logicznej jak XOR (Exclusive-OR):



Rozczarowanie sieciami neuronowymi, po pierwszej fali entuzjazmu, było tak głębokie, że sieci neuronowe przestały być modnym tematem prac naukowych. Świat naukowy przez ponad 10 lat (1970-1986) wręcz ignorował pojawiające się niezależne wyniki z algorytmem propagacji wstecznej.

Historia ANN: druga rewolucja

Dopiero w 1986 praca *“Learning representations by back-propagating errors”* (Rumelhart, Hinton, Williams) spowodowała ponowne zainteresowanie, a wręcz drugą rewolucję sztucznych sieci neuronowych. Algorytm wstecznej propagacji błędów pozwalał skutecznie uczyć wielowarstwowe sieci neuronowe, przewyżając problem funkcji XOR, i wiele innych barier.

W ramach tej rewolucji powstało niewiarygodne mnóstwo publikacji, projektów, i zastosowań wykorzystujących sieci neuronowe w wielu dziedzinach nauki i techniki. Użycie hasła „sztuczne sieci neuronowe” niemal gwarantowało akceptację artykułu do publikacji, lub przyznanie finansowania projektu badawczego.

Tę rewolucję nazywano **konekcjonizmem**.

Historia ANN: drugie ochłodzenie

Pod koniec lat 90-tych XX wieku entuzjazm świata naukowego związany ze sztucznymi sieciami neuronowymi zaczął wygasać. ANN stały się ustabilizowaną technologią, ich właściwości były dobrze znane, i przestały się pojawiać nowe rewelacyjne doniesienia.

Możliwości techniczne sztucznych sieci neuronowych były limitowane przez istniejącą w tych latach technologię komputerową. Dla implementacji sieci zdolnej do choćby minimalnie skomplikowanych obliczeń potrzebna była sieć wielowarstwowa, lecz uczenie wielu warstw przy dużej liczbie neuronów było obliczeniowo niepraktyczne (tygodnie, miesiące, lub lata obliczeń). Dominował również pogląd, że możliwości sieci o większej liczbie warstw ukrytych nie przewyższają sieci z jedną warstwą ukrytą.

Należy pamiętać, że lata 90-te XX wieku są również początkowymi latami rewolucji internetowej. Rozszerzał się dostęp do sieci, oprogramowanie, i związane z nimi możliwości. **Nie był to jednak jeszcze wiek „big data” i tworzenie sieci neuronowych o dużych możliwościach było limitowane przez ilość dostępnych danych.** Zbiory danych liczyły maksymalnie tysiące próbek, i ograniczeniem jakości obliczeń sieci był kompromis pomiędzy szumami w danych, a obroną przed przeuczeniem.

Historia ANN: trzecia rewolucja — sieci głębokie

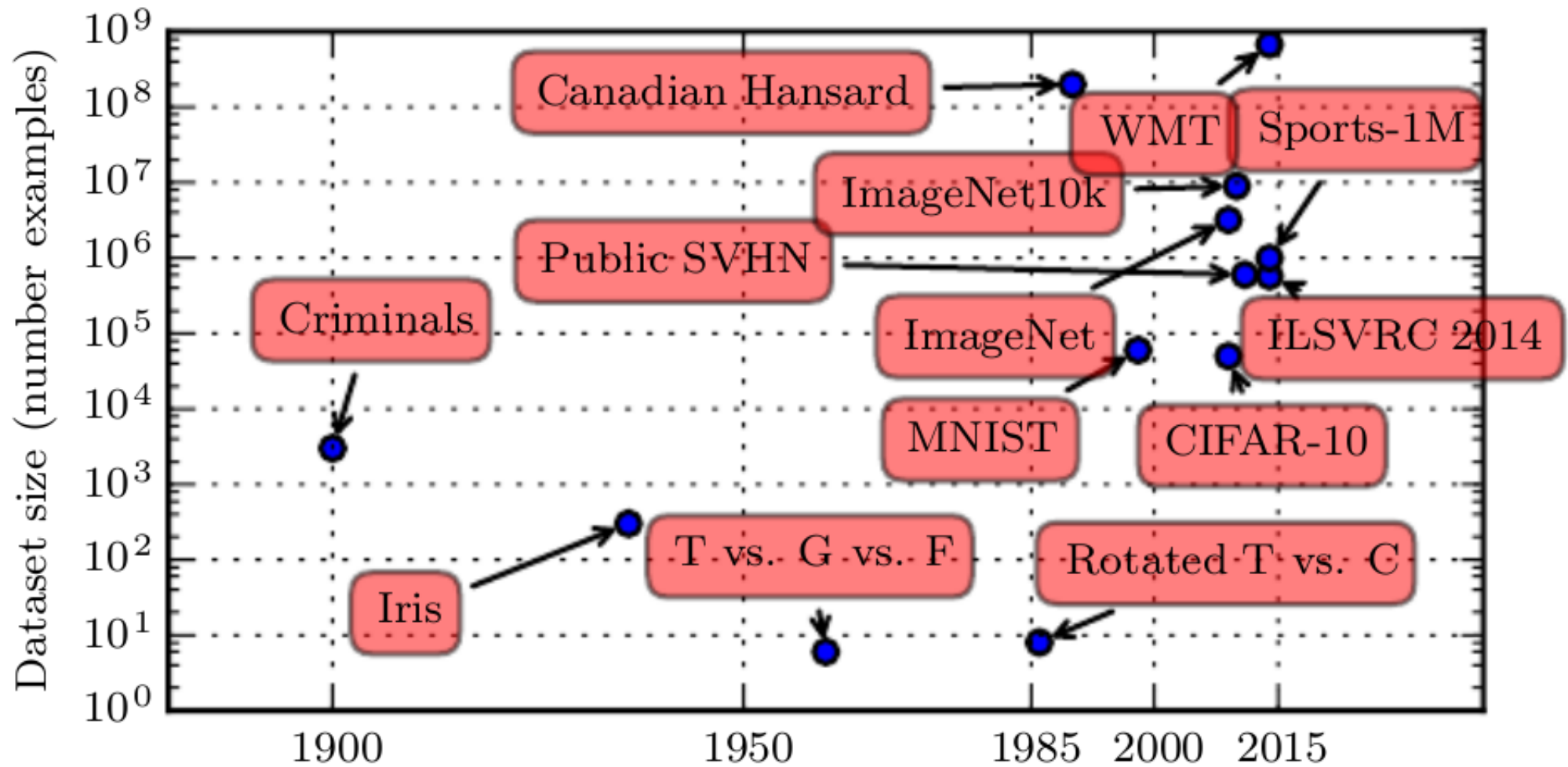
Pomimo gasnącego entuzjazmu świata naukowego sztucznymi sieciami neuronowymi, w latach 90-tych XX wieku powstało szereg nowych technologii, które miały znaleźć szersze zastosowanie dopiero w przyszłości. Przykładem jest rozproszona reprezentacja, gdzie dedykowane neurony są odpowiedzialne za kodowanie oddzielnych aspektów reprezentacji obiektu (takich jak kształt albo kolor). Innym przykładem jest sieć LSTM (*long short-term memory*) rozwiązująca problemy modelowania długich sekwencji.

Kolejny przełom rozpoczął się około roku 2006, kiedy zaczęły pojawiać się **sieci głębokie** i skuteczne metody ich uczenia. Początkowe takie sieci wykorzystywały algorytmy uczenia nienadzorowanego i demonstrowały możliwości skutecznej generalizacji na bazie niewielkich zbiorów danych.

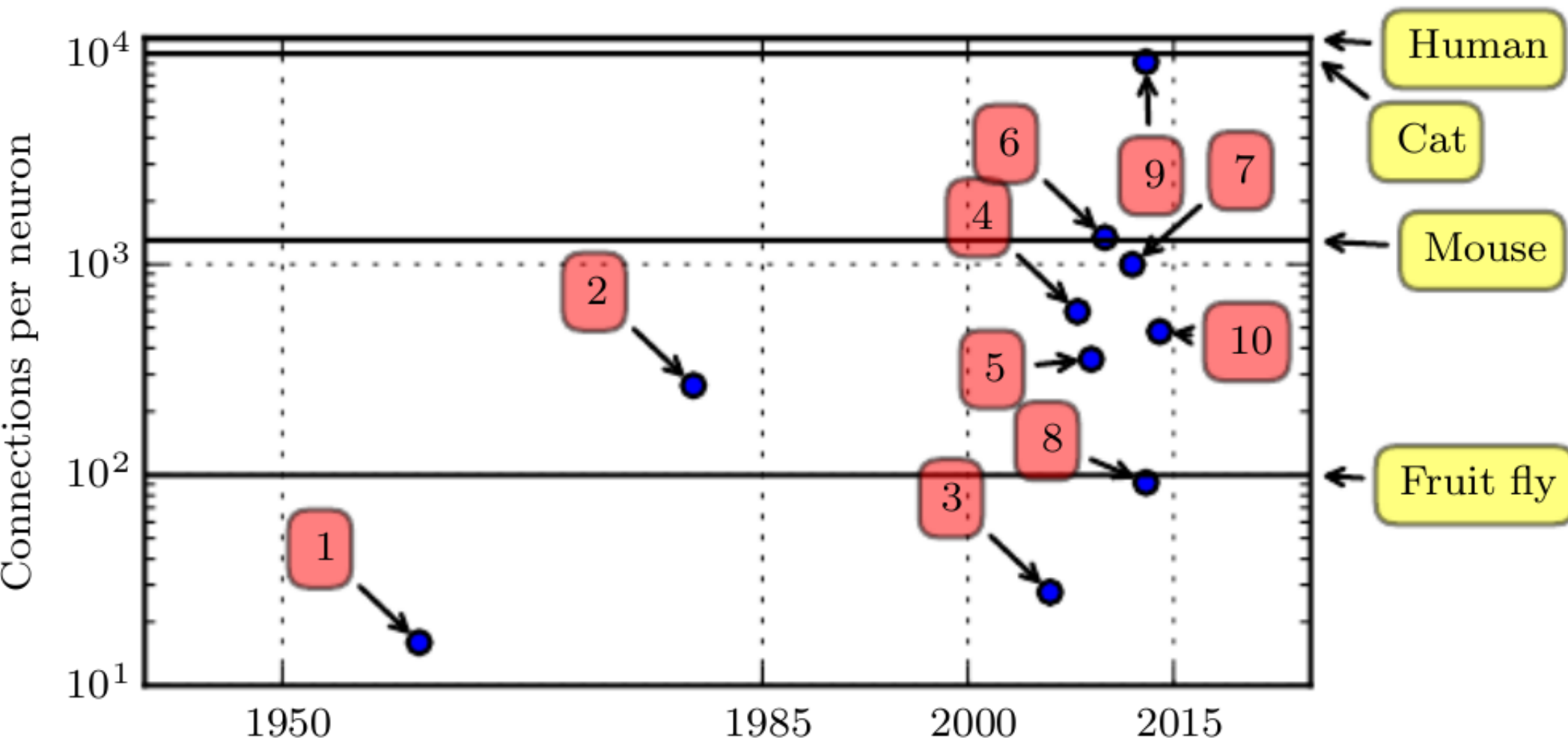
Od tego czasu **główny nurt skierował się na wykorzystanie nowych, ogromnych zbiorów danych, lecz z wykorzystaniem metod stosowanych z powodzeniem wcześniej, takich jak wielowarstwowe perceptrony i uczenie metodą wstecznej propagacji błędów.**

Należy dodać, że ten przełom stał się możliwy częściowo dzięki postępom w rozwoju technologii komputerowej. **Niemalą udział miało w nim pojawienie się elementów obliczeniowych w postaci kart graficznych GPU z tysiącami rdzeni,** które przeznaczone są do obliczania elementów obrazu, ale mogą być wykorzystane do dowolnych obliczeń.

Wzrost zbiorów danych



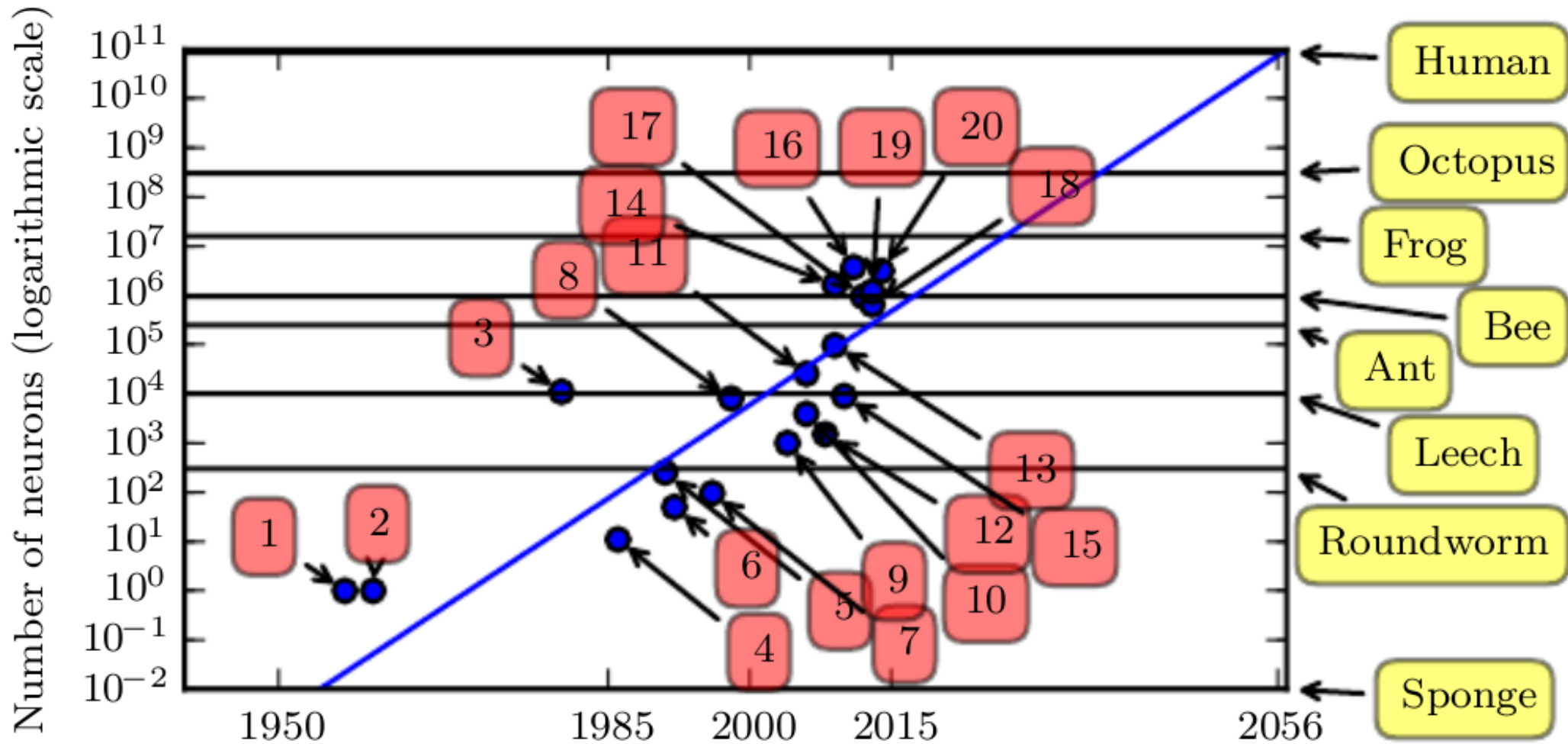
Wzrost liczby połączeń



1. Adaptive linear element (Widrow, Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla et al., 2006)
4. Deep Boltzmann machine (Salakhutdinov, Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett et al., 2009)

6. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
7. Distributed autoencoder (Le et al., 2012)
8. Multi-GPU convolutional network (Krizhevsky et al., 2012)
9. COTS HPC unsupervised convolutional network (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)

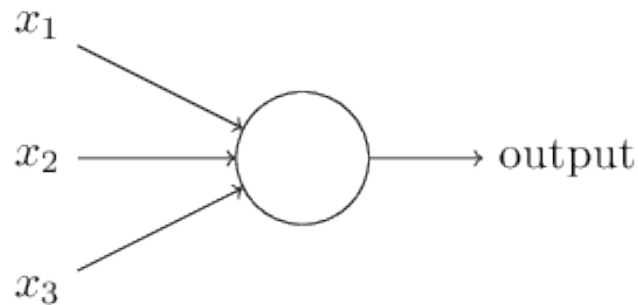
Wzrost wielkości sieci



1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow, Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart et al., 1986b)
5. Recurrent neural network for speech recognition (Robinson, Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio et al., 1991)
7. Mean field sigmoid belief network (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998b)
9. Echo state network (Jaeger, Haas, 2004)
10. Deep belief network (Hinton et al., 2006)
11. GPU-accelerated convolutional network (Chellapilla et al., 2006)
12. Deep Boltzmann machine (Salakhutdinov, Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina et al., 2009)
14. Unsupervised convolutional network (Jarrett et al., 2009)
15. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
16. OMP-1 network (Coates, Ng, 2011)
17. Distributed autoencoder (Le et al., 2012)
18. Multi-GPU convolutional network (Krizhevsky et al., 2012)
19. COTS HPC unsupervised convolutional network (Coates et al., 2013)
20. GoogLeNet (Szegedy et al., 2014a)

Perceptron

Perceptron jest jednym z pierwszych modeli neuronu zastosowanych w sztucznych sieciach neuronowych (ANN — *Artificial Neural Networks*). Perceptron jest prostym elementem obliczeniowym realizującym ważoną sumę wielu wejść połączoną z progowaniem:



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Perceptron jest najprostszym elementem decyzyjnym, obliczającym decyzje na podstawie ważonej sumy zmiennych wejściowych. Odpowiednio dobierając wagi $w_1, \dots, w_n$ i wartość progu, możemy zamodelować pewne proste schematy decyzyjne. Sygnałom wejściowym mającym duży pozytywny wpływ na pożądaną wartość wyjścia nadajemy duże wartości wag, a wejściom mającym mniejszy wpływ nadajemy odpowiednio mniejsze wagi.

Oraz, co ważniejsze, możemy nauczyć się właściwych wartości tych parametrów na podstawie odpowiedniego zbioru treningowego.

Funkcja aktywacji neuronu

Wzór roboczy aktywacji perceptronu można przepisać na nieco wygodniejszą formę:

$$\text{output} = \sigma(w^T \cdot x + b) = \begin{cases} 0 & \text{if } (w^T \cdot x + b) \leq 0 \\ 1 & \text{if } (w^T \cdot x + b) > 0 \end{cases}$$

gdzie zapis $w^T \cdot x$ oznacza iloczyn skalarny wektorów wag i sygnału wejściowego, parametr b (*bias*¹) jest negacją wartości progu, całą wartość $(w^T \cdot x + b)$ nazywa się **sumą ważoną wejść**, a funkcję σ nazywa się **funkcją aktywacji**.

Funkcję aktywacji σ zadaną powyższym wzorem nazywa się **funkcją krokową**.

Zauważmy, że parametr *bias* można potraktować jako wagę dodatkowego wejścia o stałej wartości 1, i wtedy znika on ze wzoru, kosztem bardziej rozbudowanej sieci.

Jest również możliwe budowanie sieci z wartością *biasu* równą 0, czyli praktycznie bez *biasu*, jeśli wiemy, że taka sieć prawidłowo modeluje dany proces decyzyjny.

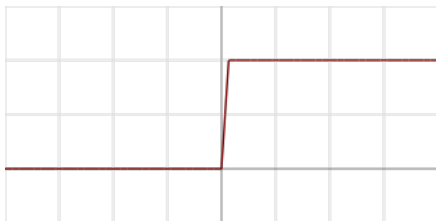
¹W polskiej literaturze sieci neuronowych termin *bias* jest czasami tłumaczony jako: próg, albo przesunięcie. Jednak większość autorów opracowań używa po prostu słowa *bias*, odmieniając go przez wszystkie polskie przypadki. Jakkolwiek ta forma początkowo trochę razi, jest jednak najwygodniejsza, dlatego będzie tu stosowana, jednakowoż pisana kursywą, aby podkreślić, że jest terminem obcym.

Sigmoidalna funkcja aktywacji

Przedstawiona krokowa funkcja aktywacji może poprawnie modelować pewne procesy podejmowania decyzji, lecz nie nadaje się do automatycznego uczenia sieci. Jeśli pewne wagi, lub *bias*, będą nieznacznie różniły się od optymalnych wartości, to możliwe są dwa przypadki. Albo perceptron da taką samą odpowiedź jak dla wartości optymalnych, i wtedy sieć będzie na pozór działać poprawnie, ale nie jest możliwe wykrycie różnicy i zrobienie poprawek. Dla danych spoza zbioru treningowego otrzymamy wtedy błędy. Albo perceptron „przeskoczy” na inną wartość wyjścia, i wtedy sieć będzie działać źle, ale nie będziemy mogli zauważyć, że wartości parametrów były bliskie poprawnych.

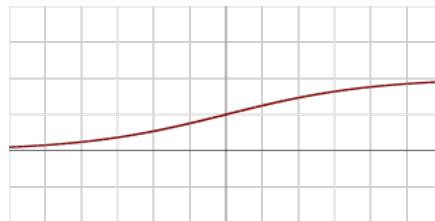
krokowa

$$\sigma(x) = \begin{cases} 0 & \text{iff } x \leq 0 \\ 1 & \text{iff } x > 0 \end{cases}$$



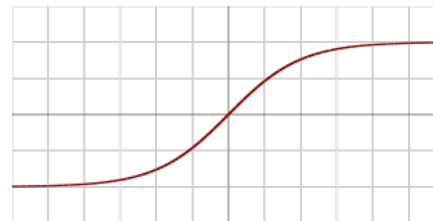
sigmoid

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



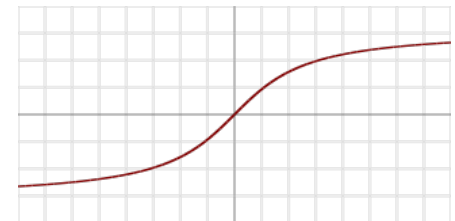
tanh

$$\begin{aligned} \sigma(x) &= \tanh(x) \\ &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \end{aligned}$$



arctan

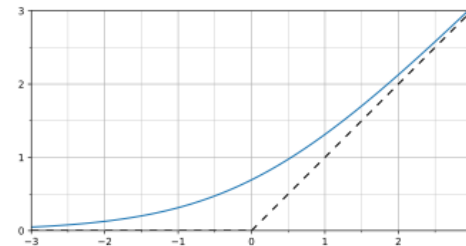
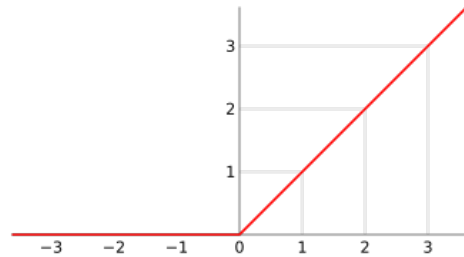
$$\sigma(x) = \tan^{-1}(x)$$



Tradycyjnie często używaną jest sigmoidalna funkcja aktywacji, zwana również krzywą logistyczną. Istnieje jednak szereg alternatywnych, również wykorzystywanych funkcji.

Alternatywne funkcje aktywacji

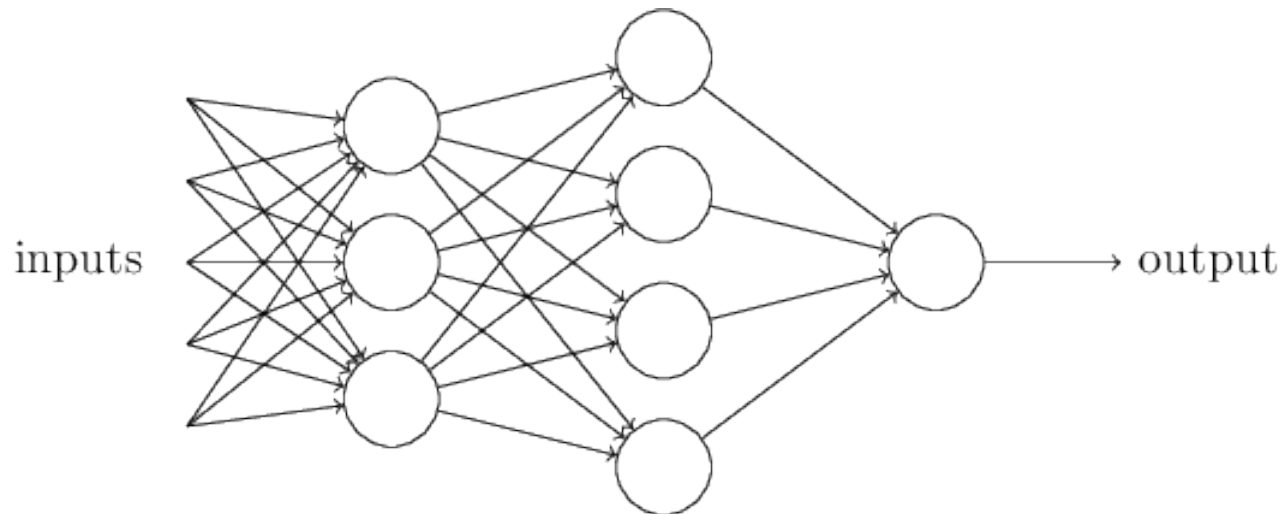
Funkcja sigmoidalna była często wykorzystywana w sztucznych sieciach neuronowych do około roku 2010. Od tego czasu dużą popularność, szczególnie w sieciach głębokich zdobywa **jednostronna funkcja prostująca** zwana również **ReLU** (*Rectified Linear Unit*), oraz szereg jej podobnych funkcji ciągle różniczkowalnych jak funkcja **softplus** dana wzorem: $f(x) = \log(1 + e^x)$:



Te funkcje aktywacji bywają używane szczególnie w zastosowaniach maszynowego widzenia, rozpoznawania mowy, i innych.

Perceptron wielowarstwowy

Pojedynczy perceptron potrafi modelować podejmowanie bardzo prostych decyzji przez obliczanie sum ważonych z progowaniem. Ponieważ jednak te obliczenia są bardzo proste, możemy wyobrazić sobie wiele takich elementów połączonych razem, w celu obliczenia bardziej złożonych decyzji. Jednym możliwym takim modelem jest **perceptron wielowarstwowy MLP** (*Multi-Layer Perceptron*).



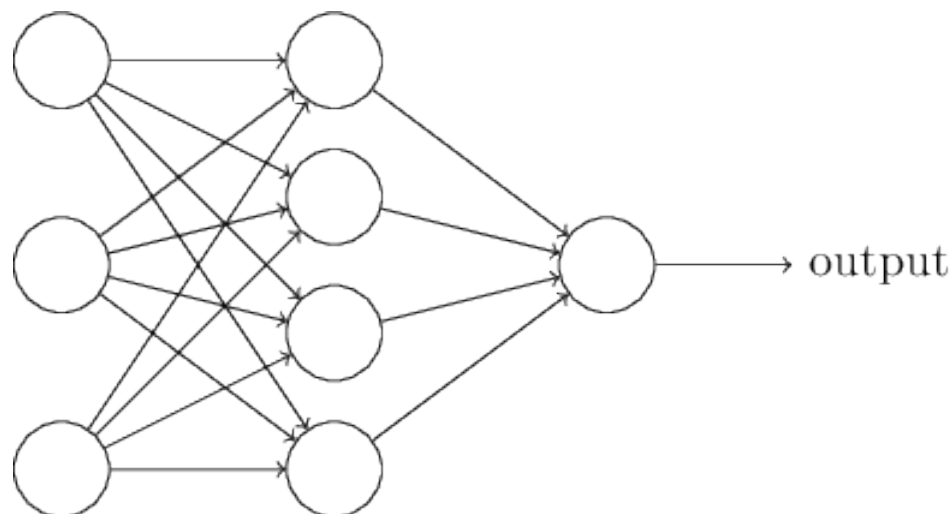
MLP składa się z szeregu warstw neuronów, z których pierwszą warstwę (do której podłączone są sygnały wejściowe) nazywamy wejściową, ostatnią warstwę (z której pobierany jest/są sygnały wyjściowe) nazywamy wyjściową, a pośrednie warstwy (których może być wiele) nazywamy warstwami ukrytymi.

Perceptron wielowarstwowy (cd.)

Dzięki warstwowej budowie, model podejmowania decyzji w MLP może być znacznie bardziej złożony. Neurony warstwy wejściowej, zamiast podejmować ostateczne decyzje, mogą rozpoznawać pewne elementarne cechy sygnału wejściowego. Analogicznie, kolejne warstwy mogą obliczać coraz to bardziej złożone wielkości, i neurony warstwy wyjściowej mogą podejmować ostateczne decyzje na podstawie znacznie bardziej istotnych kategorii niż surowe zmienne wejściowe.

Jednokierunkowe sieci neuronowe

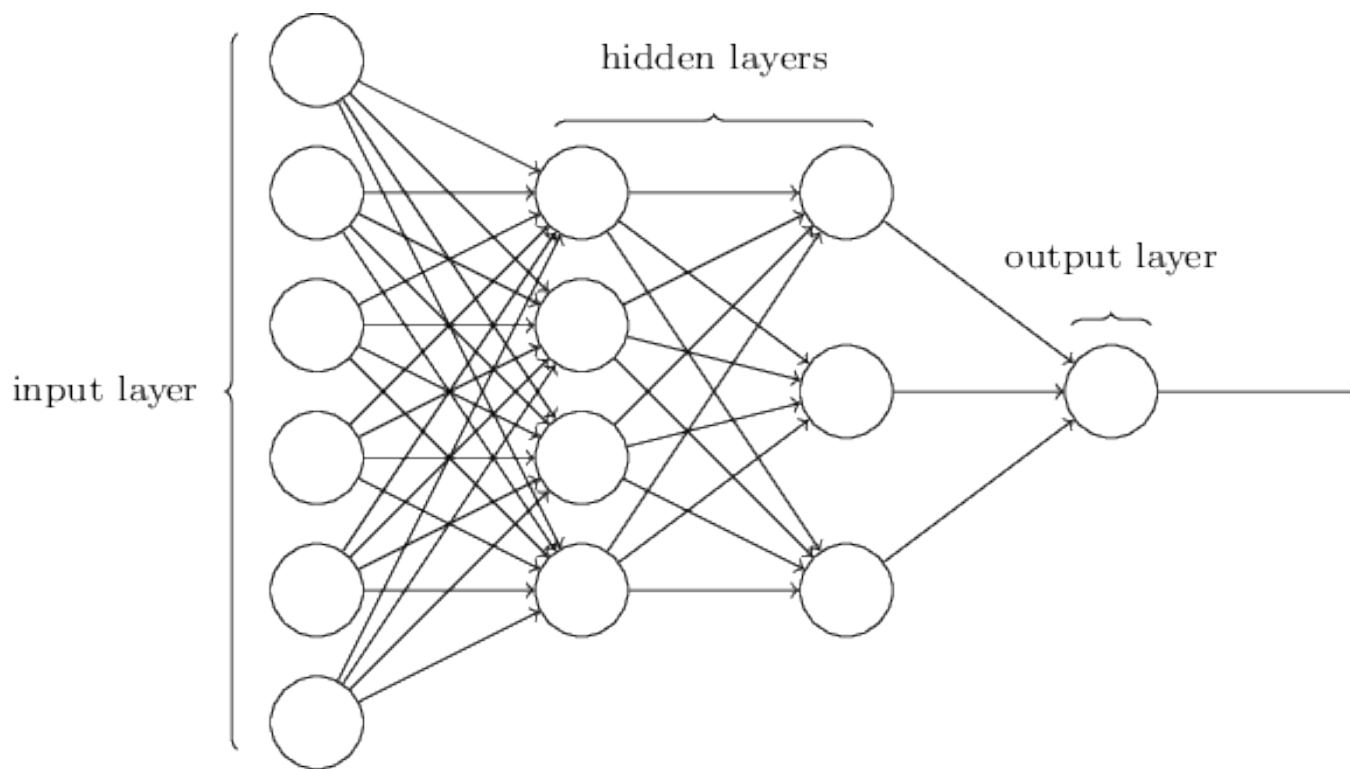
Przedstawiony wyżej perceptron wielowarstwowy MLP, ogólniej zwany **siecią jednokierunkową** (*feedforward network*), jest przykładem architektury sieci neuronowej — jedną z najpopularniejszych — lecz nie jedyną.



Taka sieć składa się z co najmniej dwóch warstw: wejściowej i wyjściowej, oraz, opcjonalnie, z pewnej liczby warstw pośrednich (na powyższym rysunku jest jedna), zwanych ukrytymi. (Nie są one ukryte w żadnym specjalnym sensie; określenie „ukryte” oznacza tylko, że nie należą one do zewnętrznego „interfejsu” sieci neuronowej.)

Architektura sieci jednokierunkowe

Architektura sieci jednokierunkowej wynika wprost z jej zastosowania. Jeśli sieć ma być użyta jako klasyfikator binarny obiektów opisanych sześcioma parametrami, to warstwa wejściowa powinna mieć sześć neuronów, pobudzanych odpowiednimi parametrami wejściowymi, a wyjście sieci będzie interpretowane jako wartość klasy obiektu.

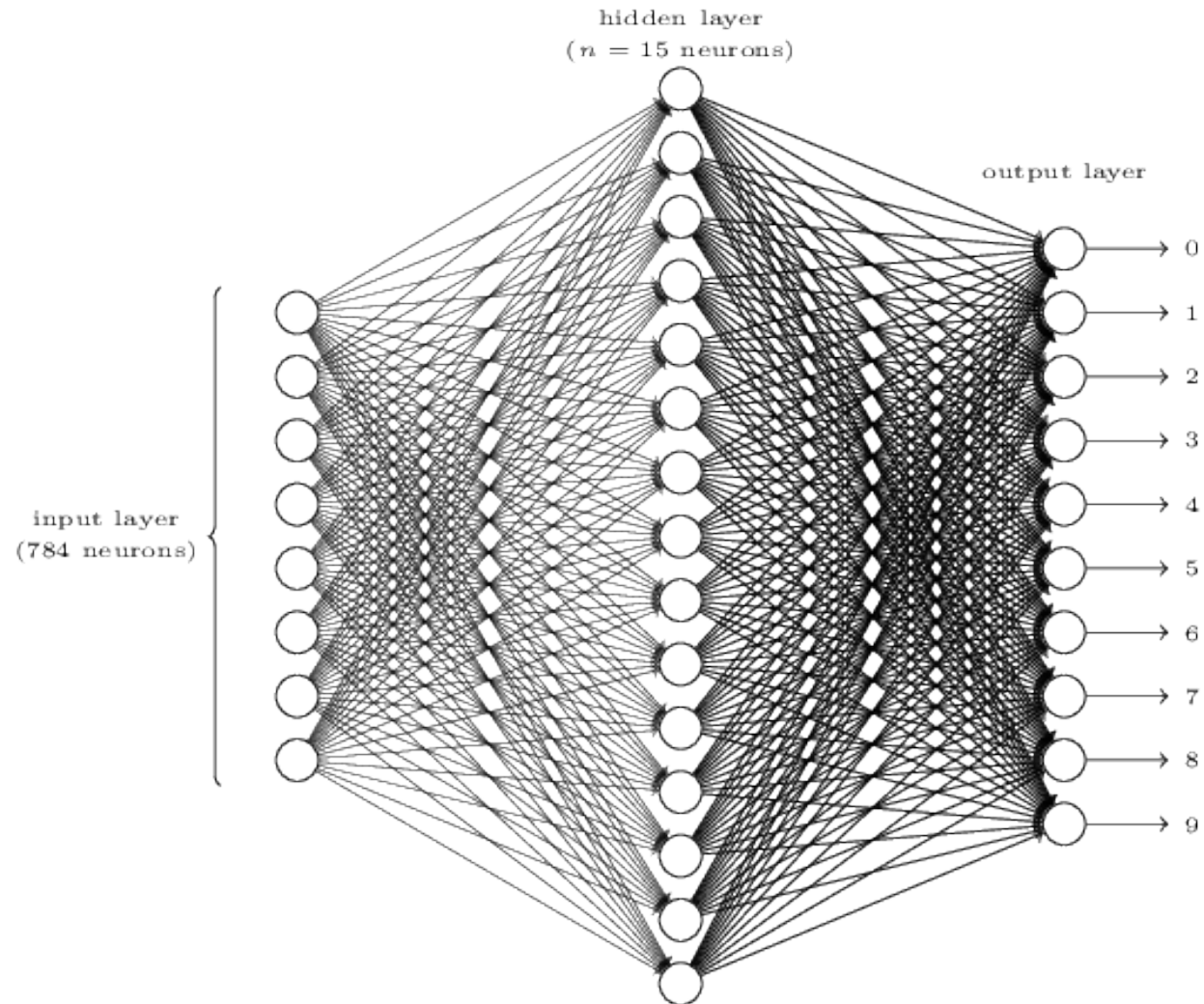


Zwróćmy uwagę, że sieć posiada wszystkie możliwe połączenia pomiędzy neuronami kolejnych warstw. Tak zwykle inicjalizuje się sieć, która potem, w trakcie uczenia, może wygasić niektóre połączenia (redukując ich wagi do bliskich zero).

Sieci jednokierunkowe: przykład

Dla odmiany rozważmy konkretną sieć przeznaczoną do rozpoznawania pisma ręcznego. Sieć ma określić, jaką cyfrę dziesiętną przedstawia posiadany obraz, zapisany w postaci rastrowej o rozmiarze 28×28 , w kodowaniu skali szarości.

Najprostszym podejściem będzie wykorzystanie wszystkich pikseli obrazu jako osobnych sygnałów wejściowych. Zatem warstwa wejściowa sieci powinna mieć $28 \times 28 = 784$ neuronów. Mogą one być pobudzane bezpośrednio wartościami odpowiednich pikseli obrazu (jasności), z pewnych względów przeskalowanymi do przedziału $[0, 1]$. Warstwa wyjściowa będzie miała dziesięć neuronów reprezentujących poszczególne cyfry, z progiem ustawionym na wartość 0.5.



Sieci jednokierunkowe: budowa warstw wewnętrznych

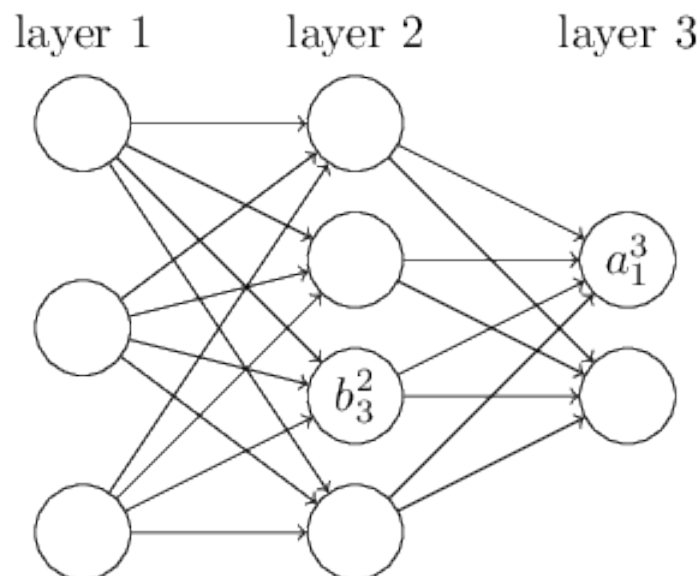
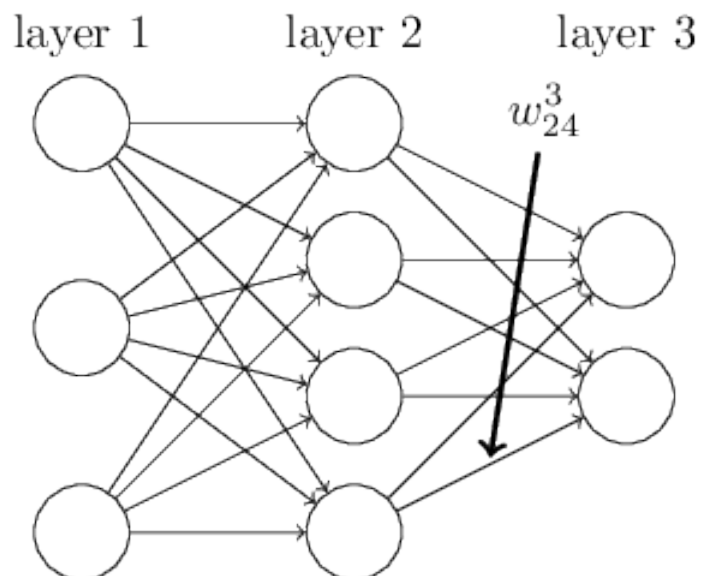
Jak już wiemy, budowa warstw wejściowej i wyjściowej, stanowiących interfejs sieci neuronowej, jest podyktowana zastosowaniem sieci i istniejącymi/wymaganymi sygnałami zewnętrznymi. Konstrukcja warstw wewnętrznych wynika z daleko bardziej złożonych rozważań. Z jednej strony, liczba neuronów i warstw wewnętrznych określa zdolność sieci do implementacji określonego schematu obliczeniowego. **Jeśli sieć będzie miała zbyt mało neuronów, to nie będzie w stanie realizować skomplikowanych procesów. Z drugiej strony, sieć o zbyt dużej liczbie neuronów będzie miała tendencję do bardzo długiego uczenia się i do przeuczenia.**

Ogólnie chcemy, by warstwa wewnętrzna miała mniej neuronów niż warstwa wejściowa, aby zmusić sieć do generalizacji — budowania bardziej zwartej reprezentacji danych wejściowych.

Osobne reguły rządzą podziałem puli neuronów wewnętrznych na warstwy. Tradycyjne podejście (do około 2010) określało liczbę warstw wewnętrznych na jedną lub dwie, w zależności od stopnia złożoności procesu obliczeniowego. Uzasadnieniem było, że większa liczba warstw, przy tej samej puli neuronów, bardzo wydłuża proces uczenia, nie zwiększając istotnie zdolności sieci do realizacji złożonych procesów obliczeniowych.

Uczenie sieci neuronowych — oznaczenia

Przyjmijmy następujące oznaczenia wag połączeń między neuronami: w_{jk}^l oznacza wagę połączenia od neuronu k -ego w $(l - 1)$ -ej warstwie do neuronu j -ego w l -tej warstwie:



Podobną notację zastosujemy dla oznaczania *biasu* i aktywacji poszczególnych neuronów: odpowiednio jako b_j^l i a_j^l oznaczmy *bias* oraz aktywację j -tego neuronu w warstwie l -tej. W tej notacji aktywacja j -tego neuronu w warstwie l dana jest wzorem:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \quad \text{w notacji algebry macierzowej:} \quad a^l = \sigma(w^l a^{l-1} + b^l)$$

Uczenie sieci neuronowych — funkcja kosztu

Algorytmy uczenia typowo dobierają wagi i wartości *biasu* w taki sposób aby osiągać minimum następującej średniokwadratowej funkcji kosztu:

$$C = \frac{1}{2n} \sum_x \| y(x) - a^L(x) \|^2$$

gdzie n jest liczbą próbek x zbioru treningowego, $y = y(x)$ jest pożądaną wartością wyjścia, L jest numerem warstwy wyjściowej (inaczej: liczbą warstw sieci), a $a^L = a^L(x)$ jest wektorem aktywacji na wyjściu sieci dla pobudzenia próbką x .

Powyższy wzór można traktować jako średnią wartość kosztu $C = \frac{1}{n} \sum_x C_x$ indywidualnych wartości funkcji kosztu kwadratowego dla pojedynczej próbki $C_x = \frac{1}{2} \| y - a^L \|^2$.

Przedstawiony poniżej algorytm **propagacji wstecznej** (*backpropagation*) oblicza błąd δ_j^l obliczeń j -tego neuronu w warstwie l -tej, oraz na tej podstawie pochodne cząstkowe $\partial C / \partial w_{jk}^l$ i $\partial C / \partial b_j^l$. W efekcie można wyznaczyć poprawki dla w_{jk}^l i b_j^l minimalizujące funkcję kosztu metodą największego gradientu. (Najczęściej uśrednia się obliczenia dla pewnej niewielkiej partii próbek co stanowi metodę stochastycznego największego gradientu.)

Uczenie sieci neuronowych — algorytm propagacji wstecznej

Krok 0 (inicjalizacja): dla danej próbki x ustaw wartości aktywacji warstwy wejściowej a^1 .

Krok 1 (propagacja): dla kolejnych warstw $l = 2, 3, \dots, L$ oblicz $z^l = w^l a^{l-1} + b^l$ i $a^l = \sigma(z^l)$.

Krok 2 (błąd wyjścia): oblicz wektor błędów $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
(symbol $\odot$ oznacza iloczyn Hadamarda, dostępny w Matlabie jako `.*`)

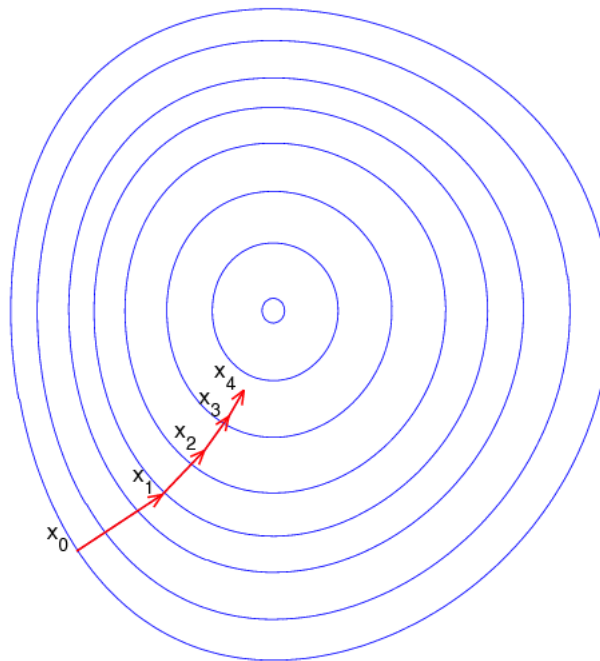
Krok 4 (propagacja wsteczna): dla kolejnych warstw $l = L, L - 1, \dots, 3, 2, 1$ oblicz wektory błędów $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

Krok 5 (poprawki): gradient funkcji kosztu jest dany jako $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ i $\frac{\partial C}{\partial b_j^l} = \delta_j^l$ zatem jako poprawki do wag w_{jk}^l i wartości biasu b_j^l należy zastosować negacje tych gradientów, z pewnym małym współczynnikiem uczenia η , zapewniającym stabilne uczenie (odporność na szumy w danych treningowych).

Uzupełnienia: wzory poprawek dla mini-batch

Poszukiwanie rozwiązania metodą gradientu

Algorytm propagacji wstecznej poszukuje rozwiązania za pomocą gradientu funkcji kosztu. W najprostszym przypadku wykorzystuje on metodę **gradientu prostego** (*gradient descent*):



Często wykorzystywanym wariantem tej metody jest jej przybliżenie probabilistyczne zwane **gradientu stochastycznego SGD** (*stochastic gradient descent*), często lepiej sprawujące się w przypadku funkcji kosztu o nieregularnym kształcie.

Usprawnienia procesu uczenia

Człowiek uczy się znacznie szybciej, gdy widzi, że popełnia duże błędy. Jego reakcje są wtedy zdecydowane, i wprowadzane poprawki są znaczące. Gdy błędy, które popełnia stają się minimalne, proces uczenia spowalnia. Pytanie, czy podobne zjawisko można zaobserwować w uczeniu sztucznych sieci neuronowych.

Jak widzieliśmy w algorytmie wstecznej propagacji, poprawki wag i *biasu* są określone przez wartości $\frac{\partial C}{\partial w}$ oraz $\frac{\partial C}{\partial b}$, gdzie kwadratowa funkcja kosztu jest dana wzorem (wzory ponownie są dostosowane do przypadku, gdy pożądana jest wartość wyjścia $y = 1.0$):

$$C = \frac{(y - a(x))^2}{2}$$

gdzie $a(x) = \sigma(wx + b)$ jest wartością aktywacji neuronu.

Powyższe pochodne cząstkowe funkcji kosztu można obliczyć jako proporcjonalne do pochodnej funkcji aktywacji $a'(x)$. Zatem wielkość dokonywanych w procesie uczenia poprawek jest związana z pochodną funkcji aktywacji.

Pochodna funkcji sigmoidalnej

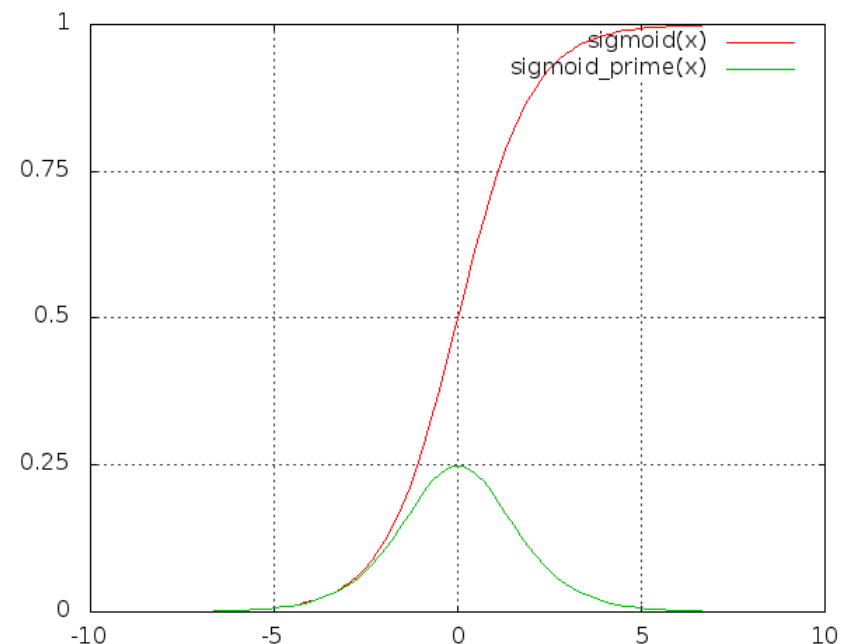
Pochodną funkcji sigmoidalnej można wyliczyć następująco. (Zwróćmy uwagę, że druga postać pochodnej ma znaczenie w procesie uczenia. Algorytm propagacji wstecznej oblicza wszystkie wartości aktywacji podczas pierwszej fazy propagacji. Mogą one być zapamiętane przez algorytm, i użyte do szybkiego obliczania wartości pochodnych w fazie propagacji wstecznej błędów.)

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

$$\sigma'(x) = \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}}\right)$$

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

```
$ gnuplot
sigmoid(x)=1/(1+exp(-x))
sigmoid_prime(x)=sigmoid(x)*(1.-sigmoid(x))
plot sigmoid(x),sigmoid_prime(x)
```



Jak widać, gdy sieć jest daleko od pożądanej wartości $y = 0.0$, funkcja sigmoidalna jest płaska, i jej pochodna ma bardzo małe wartości. Powoduje to, że uczenie postępuje bardzo powoli gdy sieć ma duże błędy, i przyspiesza dopiero w bezpośrednim sąsiedztwie wartości docelowych.

Obliczanie kosztu metodą entropii krzyżowej

Spowolnienie uczenia sieci wielowarstwowej można zaobserwować uruchamiając mały przykład:

http://neuralnetworksanddeeplearning.com/chap3.html#saturation2_anchor

Ten problem spowolnienia uczenia sieci można rozwiązać stosując, zamiast średniokwadratowej funkcji kosztu następującą funkcję **entropii krzyżowej**:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

gdzie sumowanie następuje po wszystkich n próbkach uczących $\langle x, y \rangle$.

Sensowność stosowania tej funkcji wynika z pewnych własności statystycznych, które nie będą tu omawiane. Zauważmy jedynie, że powyższa funkcja (i) jest nieujemna, (ii) zanika do zera w pobliżu poprawnego rozwiązania (przy założeniu, że pożądana wartość $y = 0.0$ i wtedy $a \approx 0.0$).

Można obliczyć, że pochodne cząstkowe $\frac{\partial C}{\partial w}$ oraz $\frac{\partial C}{\partial b}$ są w tym przypadku proporcjonalne do różnicy pomiędzy poprawną wartością wyjścia a aktywacją neuronu. Powoduje to stosowanie większych poprawek gdy sieć jest daleko od pożądanego regionu, i ich zmniejszanie w miarę zbliżania się do rozwiązania.

Wzór na sumaryczny koszt (błąd) dla wszystkich neuronów j warstwy L sieci:

$$C = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)]$$

Interpretacja łącznego wektora wartości na wyjściu sieci

Neurony warstwy wyjściowej wielowarstwowej sieci neuronowej indywidualnie obliczają swoje wartości, które łącznie stanowią odpowiedź sieci na pobudzenie wejściowe. Neurony zmieniają swoje wartości płynnie pomiędzy 0 a 1, ale sigmoidalna funkcja aktywacji w większości dąży do skrajnych wartości 0 albo 1. Możliwe jest, że oczekujemy iż jeden z neuronów zdecydowanie zbliży się do 1, a pozostałe będą dążyły do 0, albo jednocześnie wiele neuronów może zbliżać się do 1.

Pierwszy przypadek odpowiada funkcji klasyfikatora, gdy odpowiedź sieci wskazuje na jedno konkretne wyjście spośród n możliwych, natomiast drugi przypadek odpowiada dowolnej funkcji odpowiedzi sieci.

Niekiedy korzystne jest zastąpienie sigmoidalnej funkcji aktywacji inną funkcją, która wyznacza łączną odpowiedź wszystkich neuronów. Wtedy, przy zastosowaniu sieci jako klasyfikatora, wektor wartości na wyjściu sieci można traktować jako rozkład prawdopodobieństwa wyboru danego wyjścia, zamiast indywidualnego ich wskazania funkcją logistyczną.

Zastosowanie neuronów *softmax*

Opisaną wyżej charakterystykę wyjść sieci, dającą się interpretować jak rozkład prawdopodobieństwa, zapewnia zastosowanie w warstwie wyjściowej odmiennej funkcji aktywacji, zwanej *softmax*:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

Jak widać, wartość aktywacji a_j^L pojedynczego neuronu nie zależy wyłącznie od jego wartości pobudzenia z_j^L (ważonej sumy wejść), ale również od wartości pobudzeń wszystkich pozostałych neuronów. Gdy rośnie pobudzenie jednego, to zwiększa się jego aktywacja, ale jednocześnie zmniejszają się aktywacje pozostałych neuronów.

Łatwo zauważyć, że suma wartości aktywacji wszystkich neuronów wynosi 1:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1$$

Funkcja kosztu dla neuronów *softmax*

Jednak przy zastosowaniu funkcji aktywacji *softmax* właściwe jest użycie jeszcze innej funkcji kosztu zwanej funkcją **wiarygodności logarytmicznej** (dla konkretnej próbki uczącej):

$$C = -\ln a_y^L$$

Gdy sieć poprawnie rozpoznaje wartość y dla tej próbki, to wartość a_y^L będzie zbliżona do 1 i koszt zostanie wyliczony jako 0. Gdy sieć nie rozpozna poprawnie próbki, to wartość a_y^L będzie mała, a odwrotność jej logarytmu będzie dużą wartością dodatnią.

Można wykazać, że wartości pochodnych cząstkowych funkcji kosztu po wagach i *biasach* są proporcjonalne do różnicy pomiędzy poprawną a określoną przez sieć wartością wyjścia, podobnie jak w przypadku entropii krzyżowej dla funkcji logistycznej.

Zatem stosowanie wyjściowej warstwy *softmax* ma sens wraz z funkcją kosztu wiarygodności logarytmicznej, podobnie jak zwykłej sigmoidalnej funkcji aktywacji wraz z funkcją entropii krzyżowej. Zaletą pierwszej kombinacji jest możliwość interpretacji wyniku jako rozkładu prawdopodobieństwa.

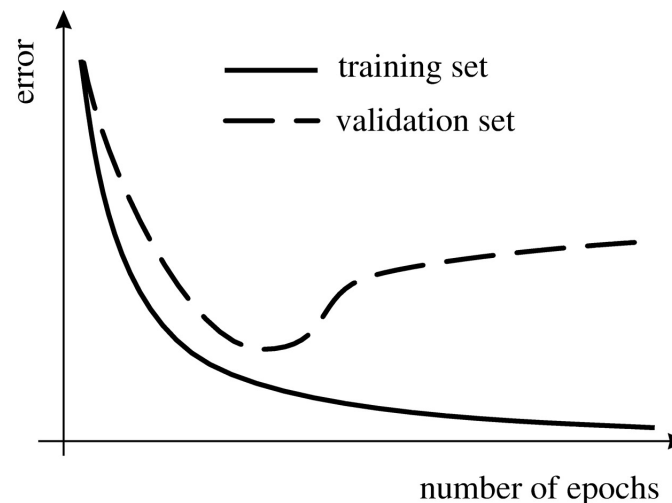
Przeuczenie w sieciach neuronowych

Sztuczna sieć neuronowa z wieloma neuronami, warstwami, i połączeniami, ma bardzo dużo parametrów. Za pomocą tych parametrów można dopasować odpowiedź sieci do pożądanej funkcji na wiele sposobów, z których tylko część (niewielka) może być poprawna. Co gorsza, **gdy w danych treningowych obecne są drobne błędy (szumy), szansa na wybranie poprawnego dopasowania zmniejsza się tym bardziej, im dokładniej uczy się sieć, ponieważ ma ona tendencję do dopasowania się do tych szumów.**

To jest problem nadmiernego dopasowania (*overfitting*), który widzieliśmy już w różnych scenariuszach, i który jest jednym z głównych problemów maszynowego uczenia.

Regularyzacja

Standardową metodą unikania przeuczenia jest stosowanie zbiorów walidacyjnych,² i zatrzymywanie uczenia gdy błędy na zbiorze walidacyjnym zaczynają rosnąć, podczas gdy błędy na zbiorze uczącym nadal się zmniejszają.



Jest to jednak metoda wejściowo-wyjściowa, traktująca algorytm uczenia jak czarną skrzynkę. Niekiedy jest możliwe wbudowanie techniki unikania przeuczenia w algorytm uczenia. Taką techniką jest **regularyzacja**, która polega na takiej modyfikacji funkcji kosztu, aby zmusić sieć do uczenia się w bardziej pożądanym sposobie (aby funkcja sieci była bardziej regularna).

²Oczywiście, inną standardową metodą unikania lub ograniczenia przeuczenia jest zwiększenie zbioru treningowego.

Regularyzacja L_2

Jedna z najczęściej stosowanych form regularyzacji, zwana regularyzacją L_2 , polega na dodaniu dodatniego składnika do funkcji kosztu według schematu:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

gdzie C_0 jest oryginalną, nieregularyzowaną funkcją kosztu.

Celem tego zabiegu jest nakłonienie sieci do preferowania niewielkich wartości wag. Duże wagi będą mogły się pojawić jedynie w przypadku, gdy jednocześnie istotnie zmniejszy się zasadnicza część funkcji kosztu (nieregularyzowana). Parametr λ służy do zarządzania kompromisem pomiędzy tendencją do zminimalizowania zasadniczej funkcji kosztu, i uzyskania niewielkich wartości wag.

Nieformalnym uzasadnieniem regularyzacji jest zwykle zasada brzytwy Ockhama: model z mniejszymi wagami uważa się za prostszy, i jeśli działa on równie dobrze jak bardziej złożony model, to zwykle będzie bardziej poprawny. W rzeczywistości jednak poprawne zastosowanie regularyzacji wymaga eksperymentowania, w szczególności ze współczynnikiem λ .

Uwagi o regularyzacji

Efektem regularyzacji jest zwykle:

- unikanie przeuczenia,
- szybsza zbieżność do większej dokładności obliczeń,
- mniejsza czułość na wybór wartości początkowych i unikanie minimów lokalnych funkcji kosztu.

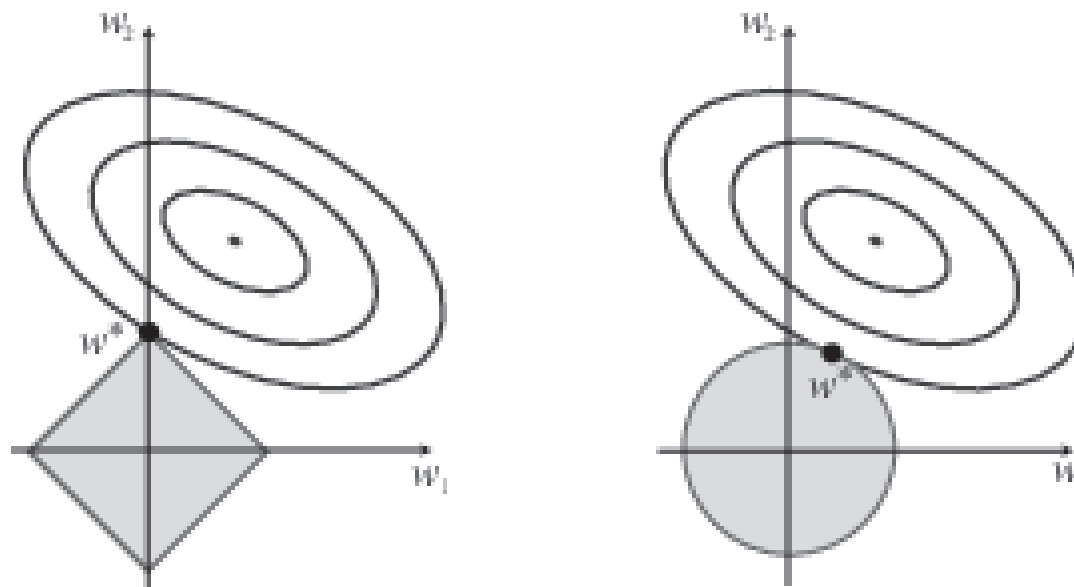
Zauważmy, że czynnik regularyzacji w podanym wyżej wzorze uwzględnia wagi, lecz nie uwzględnia *biasu*. Zatem wartości *biasu*, odmiennie niż wagi, mogą przybierać wielkie wartości. Zwykle nie stanowi to problemu, i z tego powodu nie uwzględnia się *biasu* we wzorze na regularyzację.

Regularyzacja L_1

Alternatywnym podejściem jest zastosowanie tak zwanej regularyzacji L_1 według wzoru:

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|$$

W ogólności regularyzacja L_1 działa podobnie do L_2 , wymuszając poszukiwanie mniejszych wartości wag. Jednak L_1 posiada pewne właściwości, odmienne od L_2 . Z regularyzacją L_1 częstym wynikiem jest model z wieloma wagami równymi/bliskimi 0. Tę własność można wytłumaczyć na poniższym diagramie. Ponieważ przestrzeń poszukiwania wag dla L_1 ma kształt skośnego kwadratu, punkt styku z minimum funkcji kosztu często znajdzie się na którejś z osi współrzędnych.



Regularyzacja L_1 i L_2

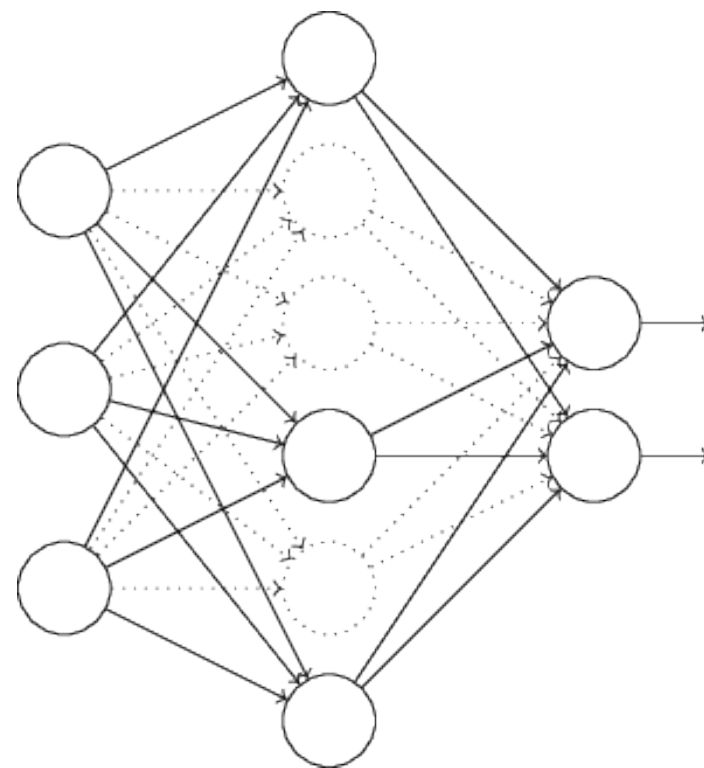
Uwaga terminologiczna: w literaturze maszynowego uczenia oznaczenia L_1 i L_2 stosowane są często zarówno do oznaczenia funkcji kosztu wartości bezwzględnej i średniokwadratowej, jak i wartości bezwzględnej i średniokwadratowej regularyzacji. Kontekst stosowania tych dwóch systemów oznaczeń jest bliski, ale oznaczają różne rzeczy.

Metoda *dropout*

Dropout jest metodą regularyzacji zupełnie odmienną od poprzednich. Polega na wielokrotnym usuwaniu losowo wybranych neuronów w trakcie uczenia (tymczasowo). Po zakończeniu fazy uczenia z usuniętymi neuronami, są one przywracane, następnie inny losowy zestaw neuronów usuwany, i uczenie kontynuowane. Ostatecznie wytrenowana sieć pracuje z wszystkimi oryginalnymi neuronami.

Na przykład, usuwając za każdym razem połowę neuronów z wybranej warstwy ukrytej dla potrzeb uczenia, w czasie rzeczywistej pracy sieci będziemy mieli czynnych dwa razy więcej neuronów niż było w czasie uczenia.

Można to skompensować dzieląc wszystkie wyuczone wagi tej warstwy przez dwa.



Procedura *dropout* jest metodą heurystyczną i podobnie jak inne podobne metody wymaga testowania szczegółów jej zastosowania. Jednak ogólnie powoduje ona efekty podobne do innych metod regularyzacji.

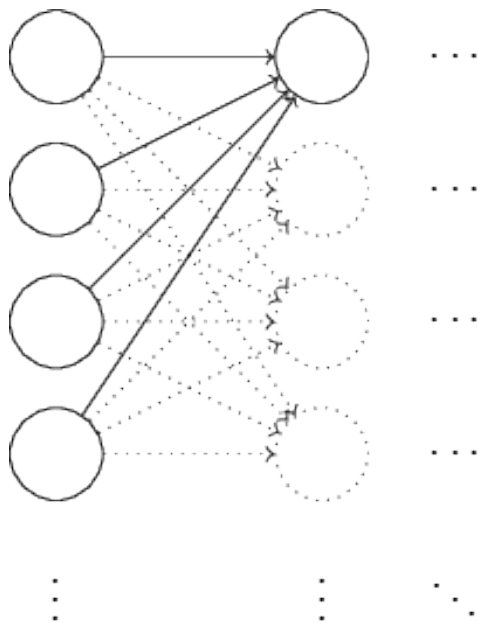
Istnieje szereg interpretacji tego efektu. Podstawowa interpretacja zakłada, że poszczególne neurony muszą „uczyć się” bardziej niezawodnych funkcji aktywacji (wag i *biasu*), ponieważ nie mogą liczyć na wspomaganie ze strony pozostałych neuronów.

Wielokrotne uczenie sieci, jakkolwiek bardziej kosztowne (bo każda faza *dropoutu* powoduje nawrót i rozpoczęcie uczenia w pewnym sensie od początku), powoduje również efekt uśredniania, który eliminuje przeuczenie, pozwala uwolnić się od lokalnych minimów funkcji kosztu, oraz uniezależnia efekt uczenia od mniej lub bardziej szczęśliwej inicjalizacji.

Inicjalizacja wag

Wybór początkowych wartości wszystkich wag i *biasów* sieci może mieć istotne znaczenie na końcowy efekt uczenia, ze względu na możliwe lokalne minima funkcji kosztu. Dlatego jako minimum stosuje się inicjalizację wartościami wybranymi losowo. Pytanie jednak, według jakiej dystrybucji ma być to losowanie. W braku lepszego pomysłu, domyślną metodą może być rozkład normalny ze średnią 0, i odchyleniem standardowym równym 1.

Okazuje się jednak, że nie jest to wcale metoda optymalna, i nawet pobieżna analiza pozwala znaleźć lepszą metodę inicjalizacji wag i *biasów*.



Problem powstaje, gdy w pewnej warstwie jest bardzo dużo neuronów (typowo tak może być w warstwie wejściowej). Przy początkowej konfiguracji, z pełnymi połączeniami, na wejściu neuronów kolejnej (drugiej) warstwy aktywacja jest sumą dużej liczby składników, każdy o wartości wylosowanej z rozkładem normalnym. Sumy ważone (pobudzenia) takich neuronów będą z reguły dużymi wartościami i neurony te będą przez początkowy czas uczenia nasycone.

Jest to to samo zjawisko początkowego spowolnienia uczenia, które już wcześniej zostało rozwiązane przez wymianę funkcji kosztu, ale tylko dla warstwy wyjściowej.

Przykładowo, jeśli w warstwie wejściowej będzie tysiąc neuronów, to można obliczyć, że przy oczekiwanej połowie wartości wejściowych równej 1 (a drugiej połowie równej 0), pobudzenie neuronu warstwy drugiej będzie miało rozkład normalny ze średnią zero i odchyleniem standardowym równym (z 500 wejść i *biasu*) $\sqrt{501} \approx 22.4$. Z dużym prawdopodobieństwem będzie to liczba o wartości bezwzględnej na tyle dużej, że aktywacja tego (każdego) neuronu drugiej warstwy będzie bardzo bliska 0 lub 1.

Można albo pozostawić korektę tej sytuacji procesowi uczenia i ... czekać, albo można dostosować do niej początkowe wagi połączeń.

W praktyce dobrym rozwiązaniem jest inicjalizacja wag połączeń każdego neuronu z n_{in} sygnałami wejściowymi rozkładem normalnym ze średnią 0 i odchyleniem standardowym równym $1/\sqrt{n_{in}}$. W większości przypadków powoduje to szybsze początkowe uczenie sieci. W niektórych przypadkach daje również lepszy wynik końcowy uczenia (lepszą generalizację, czyli niższe błędy na zbiorze testowym).

Inicjalizacja wartości *biasów* nie ma takiego wpływu na szybkość uczenia, i stosuje się albo ich inicjalizację z odchyleniem równym 1, albo nawet inicjalizację *biasów* wartością zero.

Parametry w uczeniu sieci neuronowych

Głównymi parametrami sieci neuronowej są wagi i *biasy* połączeń między neuronami.

Jednak aby zbudować sieć i wystartować proces uczenia, szereg innych parametrów musi być ustawionych, takich jak: liczba i rodzaj neuronów, liczba warstw i podział neuronów między warstwami ukrytymi, oraz różne parametry procesu uczenia, takie jak współczynnik uczenia η , współczynnik regularyzacji λ , wielkość mini-serii (*mini-batch*) m , itp. Te parametry nazywa się czasami **hiper-parametrami** (*hyper-parameters*).

Podstawową trudnością w budowie i trenowaniu sieci neuronowych jest, że wszystkie te parametry wpływają łącznie na zdolność sieci do uczenia się, wykonywania zadanych obliczeń, i osiągnięcia pożądanej dokładności i odporności na przeuczenie. Nie ma żadnych „mocnych” metod pozwalających właściwie ustawiać wartości wszystkich tych parametrów, i dla wielu z nich konieczne jest eksperymentowanie dla wyznaczenia ich optymalnych wartości. Niestety, eksperymentowanie z wszystkimi tymi parametrami naraz jest zwykle niemożliwe, ponieważ sieć z szeregiem niepoprawnie ustawionych hiper-parametrów może w ogóle nie być zdolna do uczenia się, czyli zmniejszania funkcji kosztu.

Strategia prowadzenia projektu z siecią neuronową

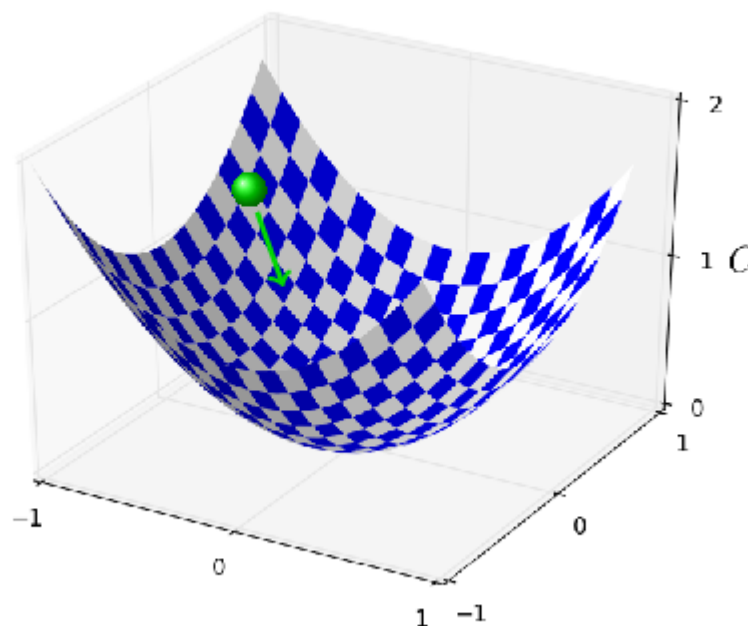
Pierwszym celem w projekcie budowy sieci neuronowej do zadanego celu powinno być osiągnięcie jakiegokolwiek nietrywialnego, czyli istotnie lepszego niż przypadkowy, efektu uczenia. Osiągnąwszy ten wynik, można przystąpić do optymalizacji poszczególnych parametrów. Jednak jego osiągnięcie bywa zadziwiająco trudne, szczególnie dla początkujących, albo w odniesieniu do nowej klasy problemów.

Warto rozpocząć projekt od zagadnienia uproszczonego do maksimum (np. ze zredukowaną liczbą klas), zbioru treningowego „oczyszczonego” z wszelkich artefaktów mogących przysparzać problemów w uczeniu sieci, wielkością i złożonością sieci w pobliżu minimum tego co spodziewamy się, że powinno przynieść efekt, i niewielkimi mini-seriami uczącymi, aby skrócić czas tych wstępnych eksperymentów, i móc łatwiej reagować na pojawiające się w wynikach trendy.

Po uzyskaniu pierwszych wskazówek, że sieć uczy się pożądanых właściwości, można zacząć eksperymenty zarówno z jej parametrami, jak i hiper-parametrami procesu uczenia, włącznie z funkcją kosztu i algorytmem uczenia.

Wartości hiper-parametrów: współczynnik uczenia η

Przypomnijmy, że współczynnik uczenia η wyznacza w algorytmie propagacji wstecznej wielkość kroku modyfikacji współczynników. Zbyt mała wartość η spowoduje bardzo powolne uczenie. Zbyt duża wartość może spowodować, że algorytm gradientowy będzie skakał wokół punktu minimum, nie mogąc w nie zejść.

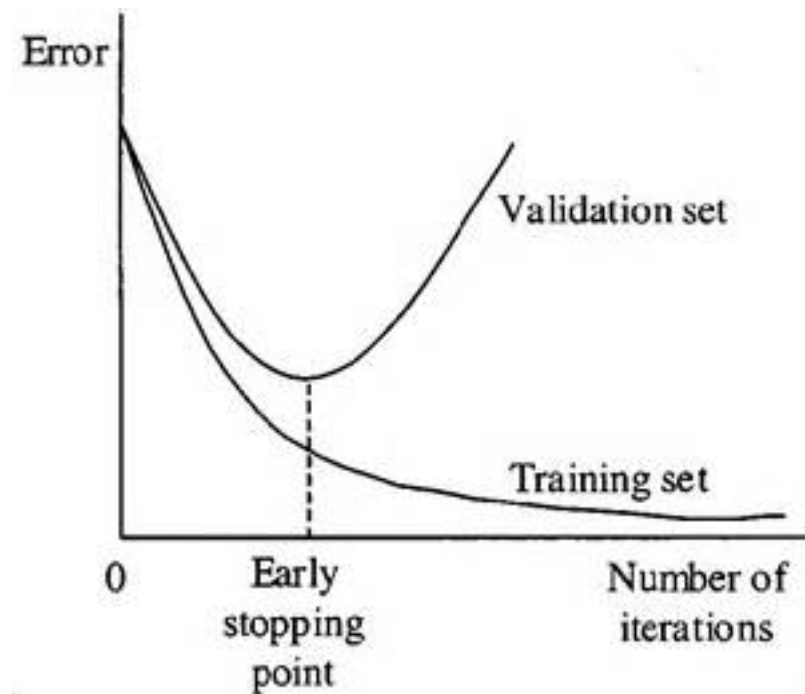


Należy zatem wyznaczyć wartość graniczną η , od której przeszukiwanie od początku poprawia kryterium kosztu. Zaczynając od jakiejś arbitralnej wartości początkowej, takiej jak $\eta = 0.01$, gdy następuje poprawa, można zwiększać η według schematu: $\eta = 0.1, 1.0, \dots$ aż do napotkania na oscylacje lub pogorszenie kryterium. Gdy od początku brak poprawy, tylko występuje pogorszenie, lub oscylacje, należy zmniejszać η według schematu: $\eta = 0.001, 0.0001, \dots$, aż do osiągnięcia poprawy.

Po określeniu wartości granicznej η należy określić wartość roboczą do optymalizacji. Powinna ona być mniejsza niż wartość graniczna, na przykład kilkukrotnie mniejsza.

Wczesne zatrzymywanie

Wczesne zatrzymywanie (*early stopping*) procesu uczenia polega na monitorowaniu postępu uczenia na zbiorze walidacyjnym. Kiedy dokładność uczenia obliczona na zbiorze walidacyjnym przestaje się poprawiać, zatrzymujemy uczenie, pomimo, iż poprawa dokładności nadal byłaby możliwa na zbiorze treningowym.



Jednak nie zawsze łatwo jest zastosować tę metodę. Wykres błędu nigdy nie jest linią ciągłą i monotoniczną, i aby zauważyć, że błędy walidacji przestały maleć, trzeba analizować go w dłuższym odcinku czasu. Zdarza się również, że błąd walidacji utrzymuje się dość długo na pewnym poziomie, a potem znowu maleje.

Zauważmy, że metoda wczesnego zatrzymywania automatycznie wyznacza **liczbę epok** uczenia.

Dostrajanie współczynnika uczenia η

Mając na uwadze metodę automatycznego zatrzymywania uczenia, możemy usprawnić nieco wcześniejszą procedurę wyznaczania współczynnika uczenia η . Zamiast utrzymywać ten współczynnik stały, można zmniejszać go w miarę postępu optymalizacji sieci.

Początkowo procedura powinna być taka jak opisana wyżej. Po zatrzymaniu uczenia współczynnik η należy zmniejszyć kilkukrotnie, i ponowić próbę uczenia. Prawie zawsze będzie możliwa dalsza poprawa dokładności, jednak o mniejszej wartości bezwzględnej, odpowiadającej szybkości uczenia.

Ten proces można powtarzać wielokrotnie, w rzeczywistości tak długo jak długo jesteśmy w stanie niezawodnie wykryć rzeczywistą poprawę kryterium uczenia.

Wartości hiper-parametrów: współczynnik regularyzacji λ

Nie ma utrwalonej metody wyboru oraz optymalizacji współczynnika regularyzacji λ . Wydaje się, że warto zaczynać bez regularyzacji ($\lambda = 0.0$) i najpierw wyznaczać współczynnik η .

Po określeniu dobrej wartości η można eksperymentować ze współczynnikiem λ zaczynając od arbitralnie wybranej wartości (np. $\lambda = 1.0$) i zwiększać lub zmniejszać tę wartość, oczekując poprawy dokładności na zbiorze walidacyjnym. Po osiągnięciu serii dobrych wartości λ można próbować dostrajać ją ze zmniejszanym stopniowo krokiem.

Po wyznaczeniu dobrej wartości λ , można ponowić strojenie wartości η .

Literatura i materiały pomocnicze

1. Stuart Russell, Peter Norvig: Sztuczna inteligencja. Nowe spojrzenie, Wydanie IV, Tom 2, Wyd.Helion, 2022, rozdział 21.
2. Michael A. Nielsen: Neural Networks and Deep Learning, Determination Press, 2015
<http://neuralnetworksanddeeplearning.com/>