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 $f(\boldsymbol{x}) = \Sigma_i \, w_i x_i$. 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 $f(\boldsymbol{x})$ 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, przezwyciężając 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 własności 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

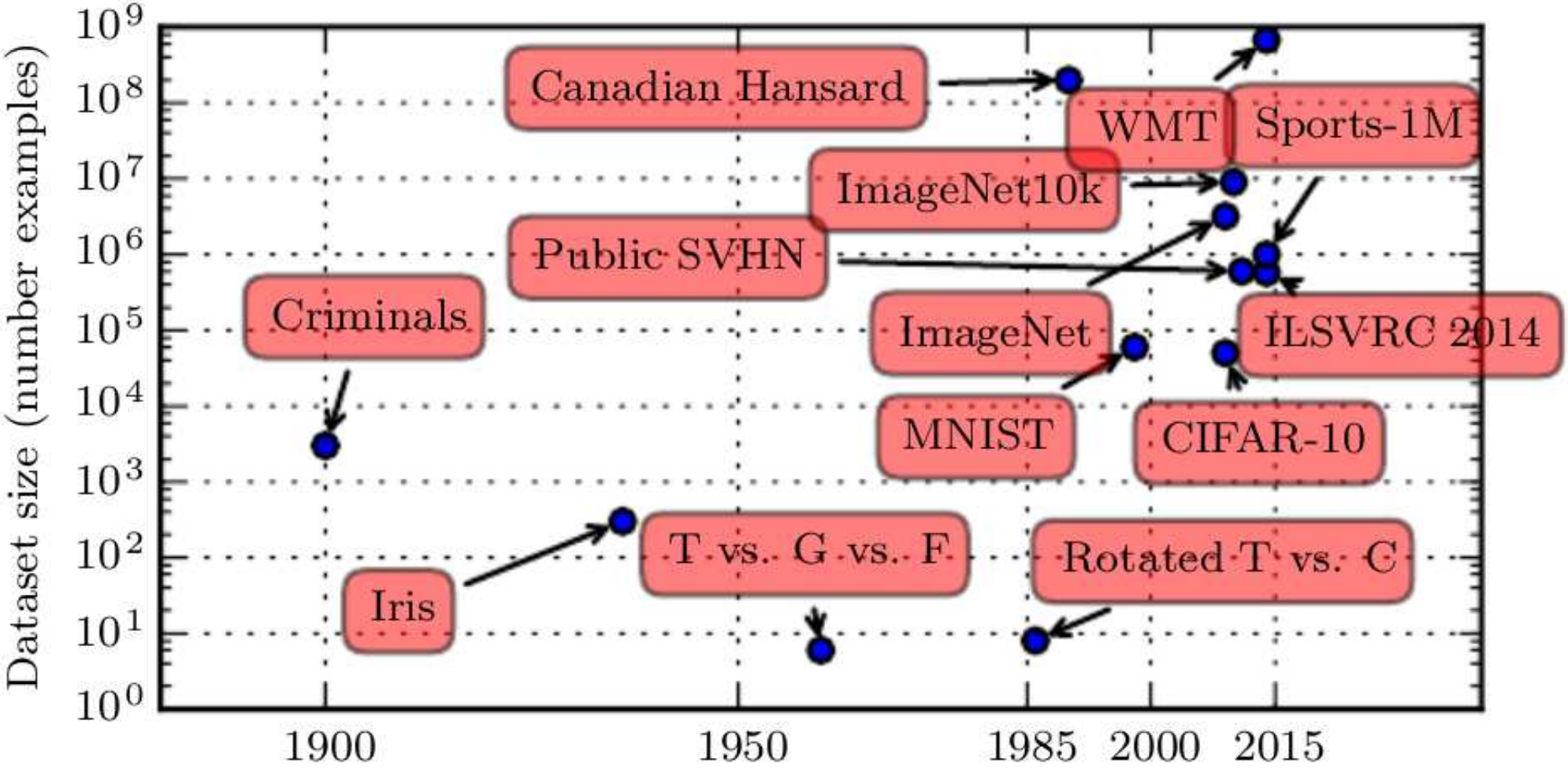
Pomimo gasnącego entuzjazmu świata naukowego sztucznymi sieciami neuronowymi, w latach 90-tych XX wieku powstało szereg nowych technologii, które miały znaleźć szersze zastosowanie dopiero w przyszłości. Przykładem jest rozproszona reprezentacja, gdzie dedykowane neurony są odpowiedzialne za kodowanie 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.

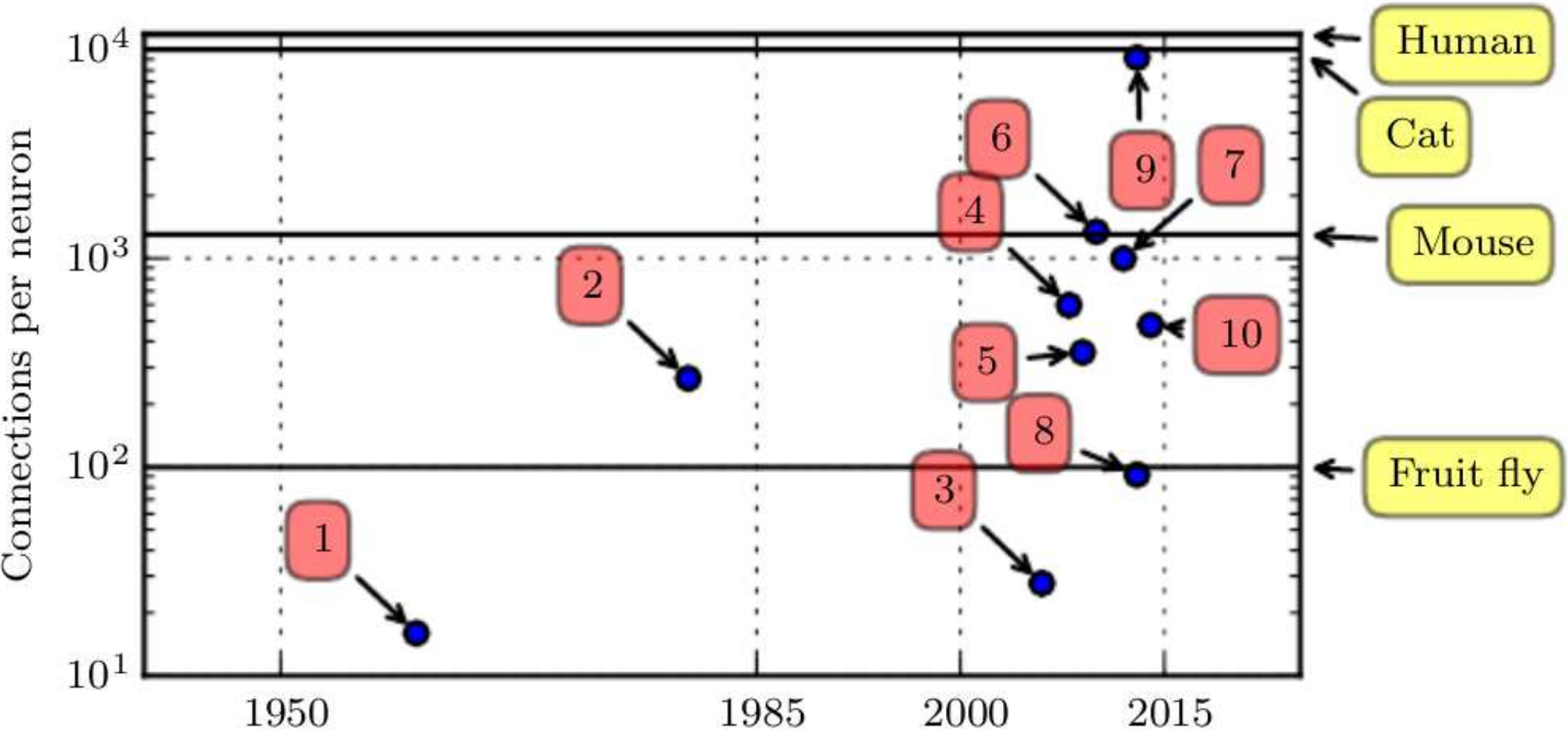
Od tego czasu główny nurt skierował się na wykorzystanie nowych, ogromnych zbiorów danych, lecz z wykorzystaniem metod stosowanych z powodzeniem wcześniej, takich jak wielowarstwowe perceptrony i uczenie metodą wstecznej propagacji błędów.

Należy dodać, że ten przełom stał się możliwy częściowo dzięki postępom w rozwoju technologii komputerowej. Niemały 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

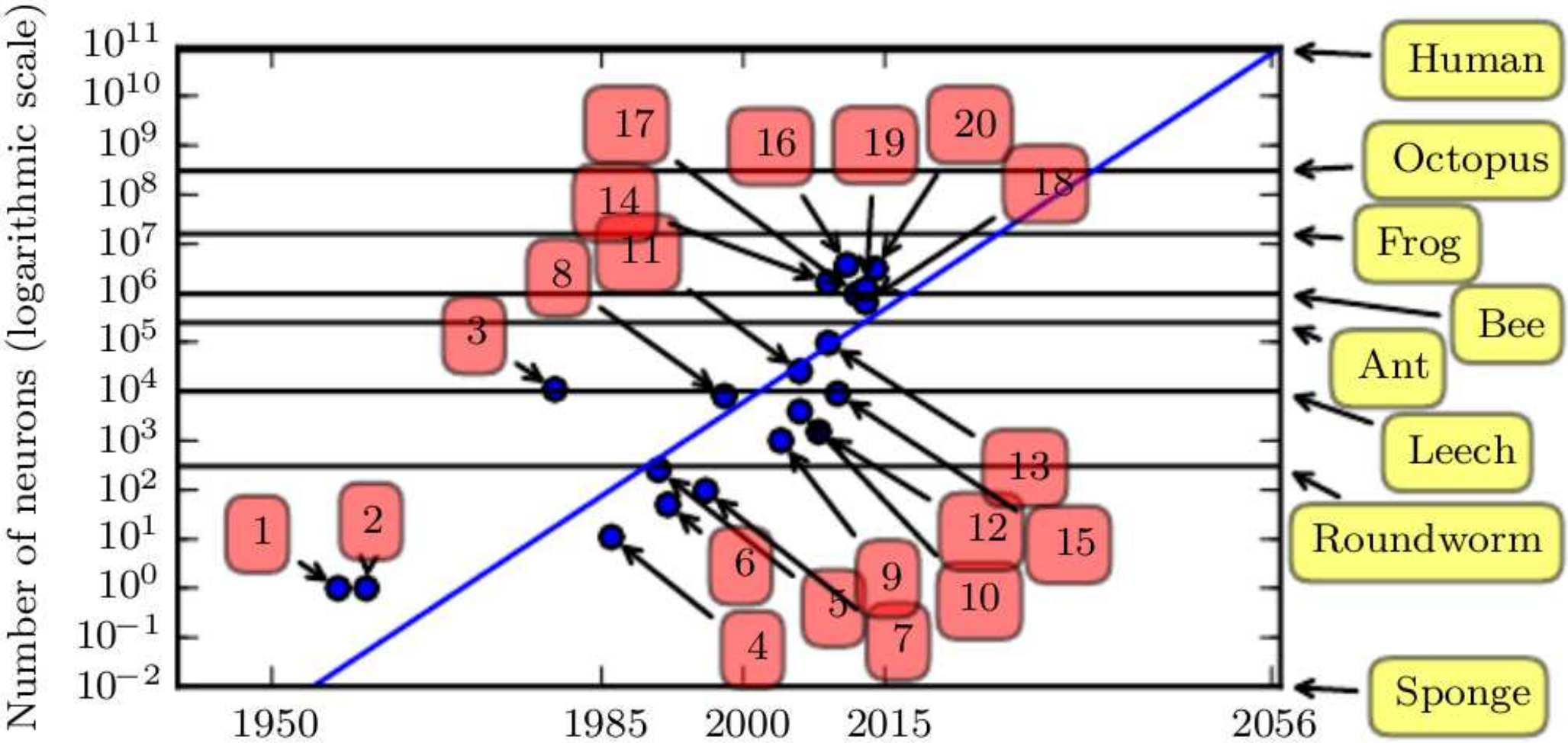


Wzrost liczby połączeń



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)

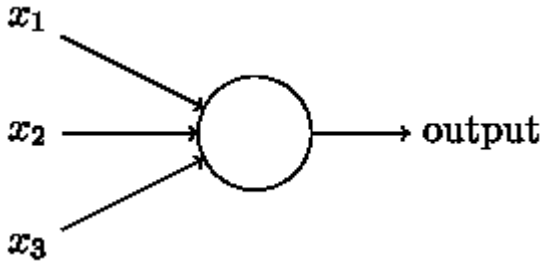
Wzrost wielkości sieci



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:



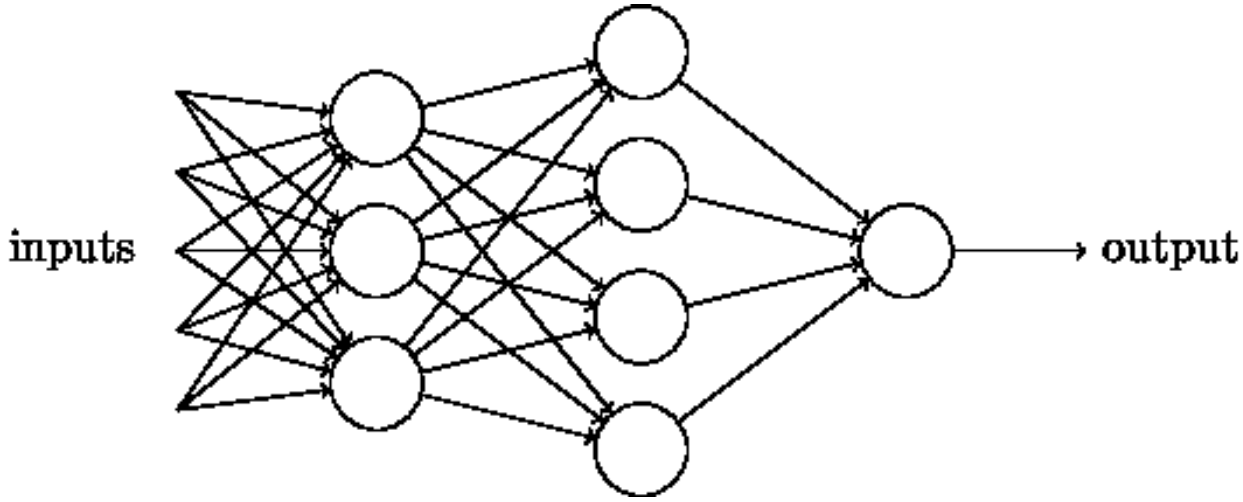
$$\text{output} = \begin{cases} 0 & \text{if } \Sigma_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \Sigma_j w_j x_j > \text{threshold} \end{cases}$$

Perceptron jest najprostszym elementem decyzyjnym, obliczającym decyzje na podstawie ważonej sumy zmiennych wejściowych. Odpowiednio dobierając wagi $w_1, ..., w_n$ 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.

Perceptron wielowarstwowy

Zanim przejdziemy do metod automatycznego uczenia sieci neuronowych, rozważmy bardziej złożony przypadek. 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.

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 odwrotnością 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 *bias*u równą 0, czyli praktycznie bez *bias*u, jeśli wiemy, że taka sieć prawidłowo modeluje dany proces decyzyjny.

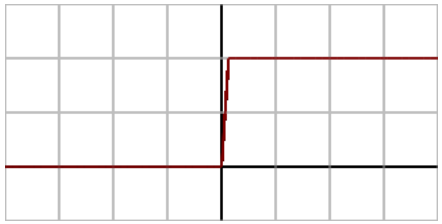
[1] W polskiej literaturze sieci neuronowych termin *bias* jest czasami tłumaczony jako: próg, albo przesunięcie. Jednak większość autorów opracowań używa po prostu słowa bias, odmieniając go przez wszystkie polskie przypadki. Jakkolwiek ta forma początkowo trochę razi, jest jednak najwygodniejsza, dlatego 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.

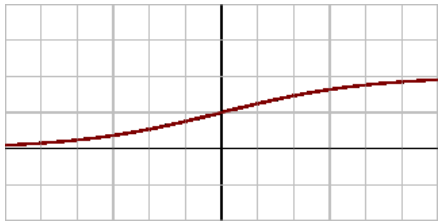
krokowa

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



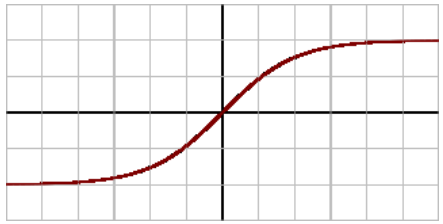
sigmoid

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



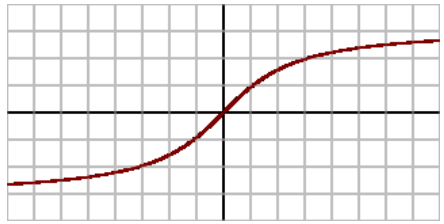
tanh

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



arctan

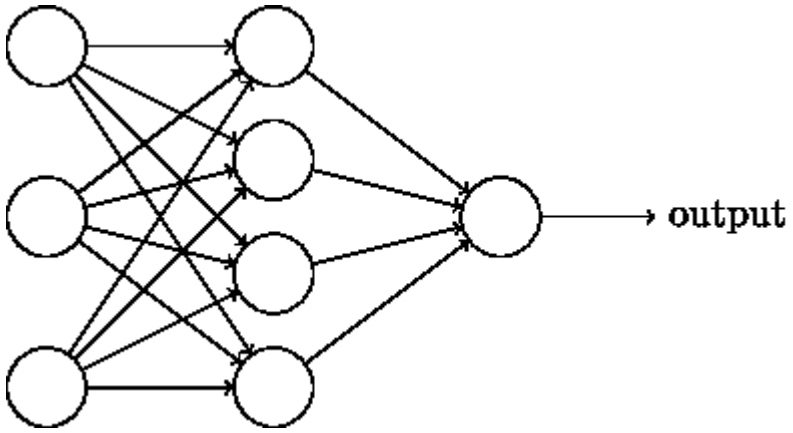
$$\sigma(x) = tan^{-1}(x)$$



Często używaną jest sigmoidalna funkcja aktywacji, zwana również krzywą logistyczną.

Architektura sieci neuronowej

Omawiany dotychczas 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.)

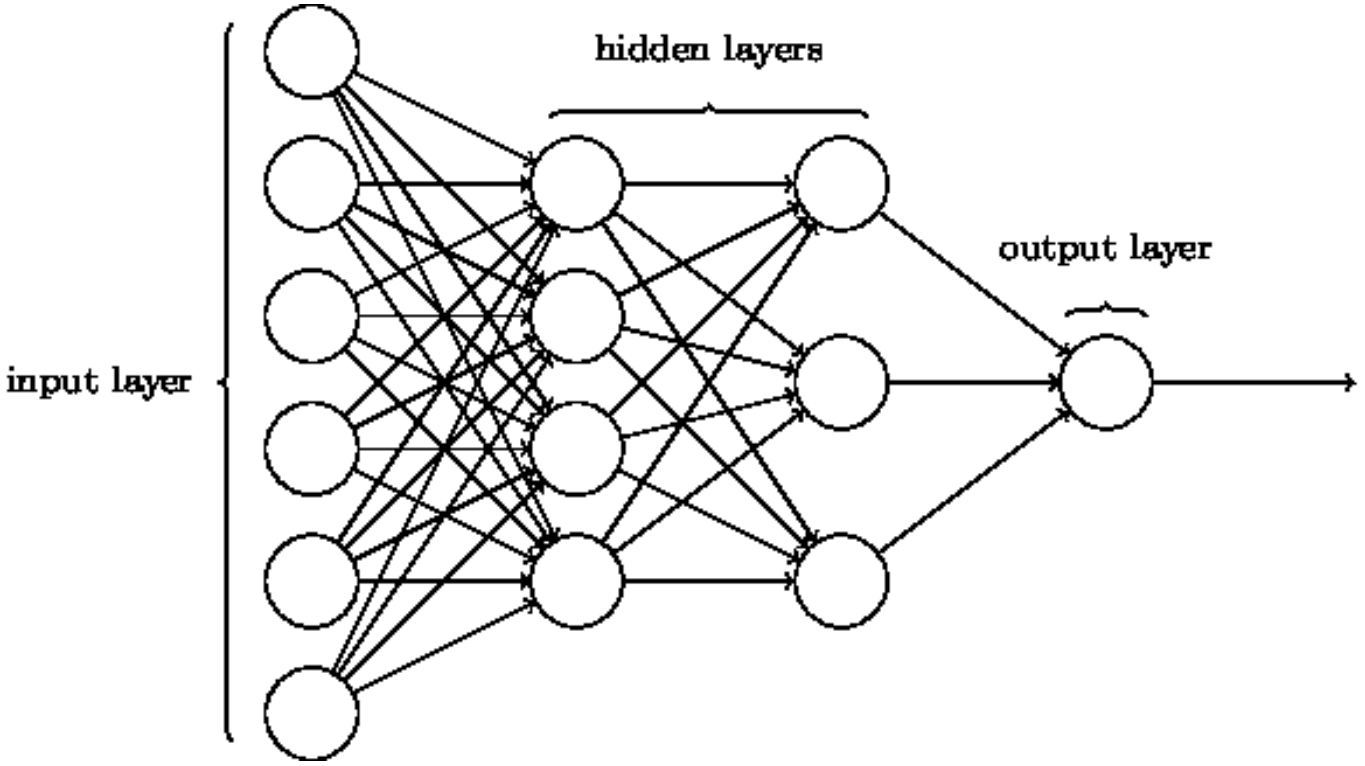
Inne architektury sieciowe

Sieci jednokierunkowe są prostym i popularnym, ale bynajmniej nie jedynym modelem sieci neuronowych. Co najmniej równie interesujące są **sieci rekurencyjne** (*recurrent networks*), w których sygnał z neuronów warstw dalszych może być przesyłany zwrotnie do warstw wcześniejszych. Ze względu na czas propagacji sygnału wynikający z niezerowego czasu pobudzenia neuronów, takie pętle nie prowadzą do nieskończonych oscylacji, tylko stwarzają możliwości realizacji zupełnie innych procesów obliczeniowych.

Uzupełnienia: przykłady sieci rekurencyjnych.

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.

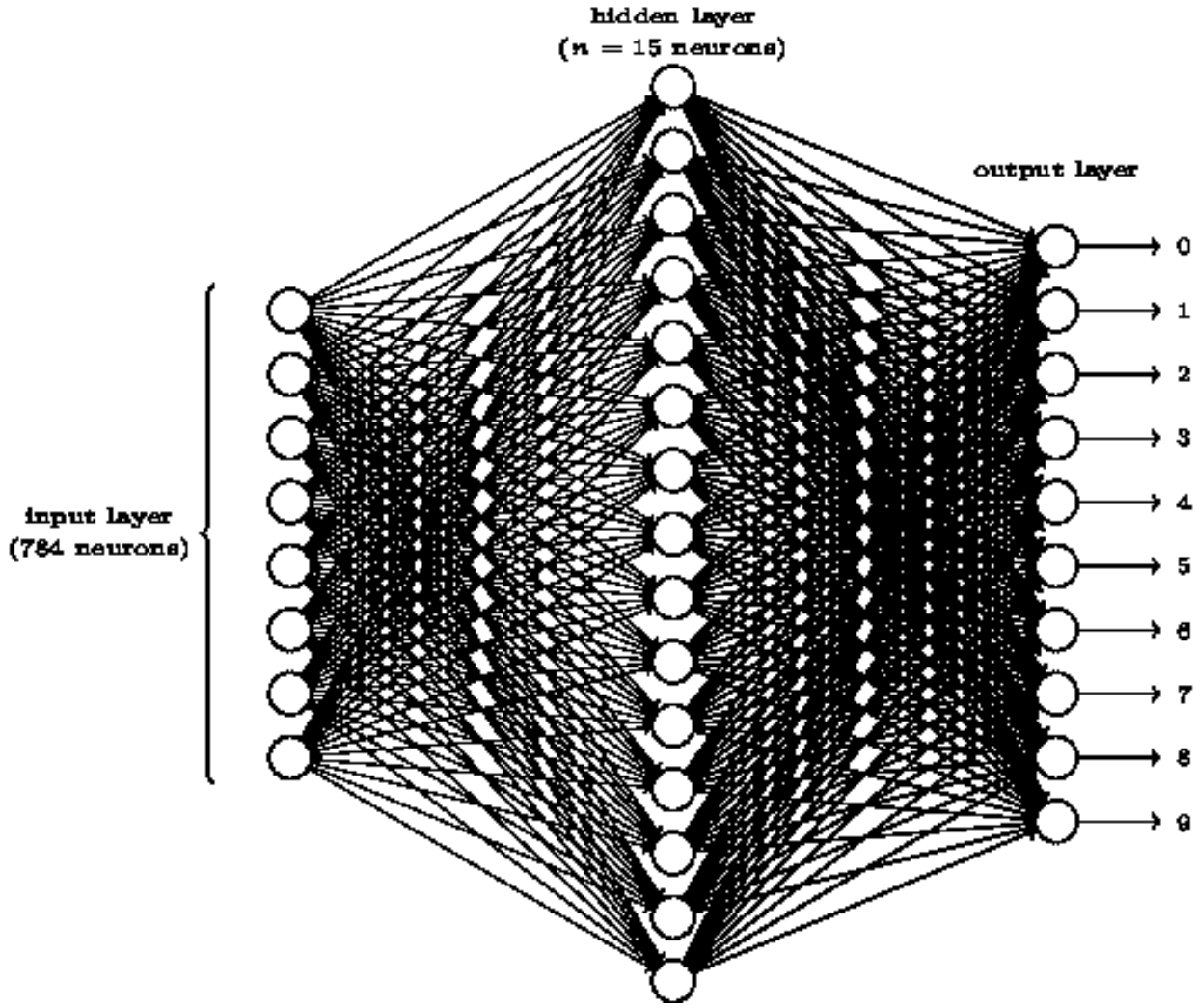


Zwróćmy uwagę, że sieć posiada wszystkie możliwe połączenia pomiędzy neuronami kolejnych warstw. Tak zwykle inicjalizuje się sieć, która potem, w trakcie uczenia, może wygasić niektóre połączenia (redukując ich wagi do bliskich zeru).

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

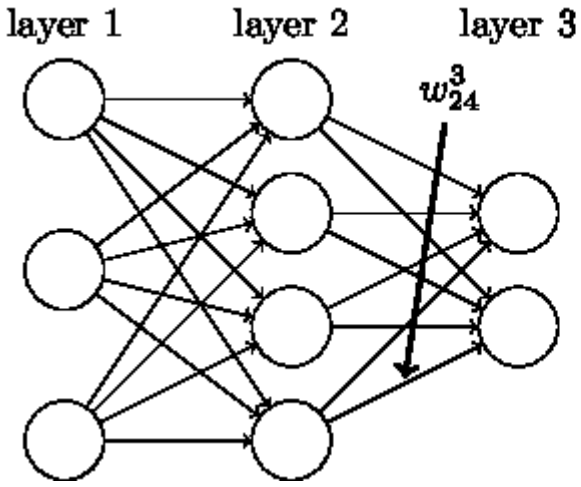
Jak już wiemy, budowa warstw wejściowej i wyjściowej, stanowiących interfejs sieci neuronowej, jest podyktowana zastosowaniem sieci i istniejącymi/wymaganymi sygnałami zewnętrznymi. Konstrukcja warstw wewnętrznych wynika z daleko bardziej złożonych rozważań. Z jednej strony, liczba neuronów i warstw 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.

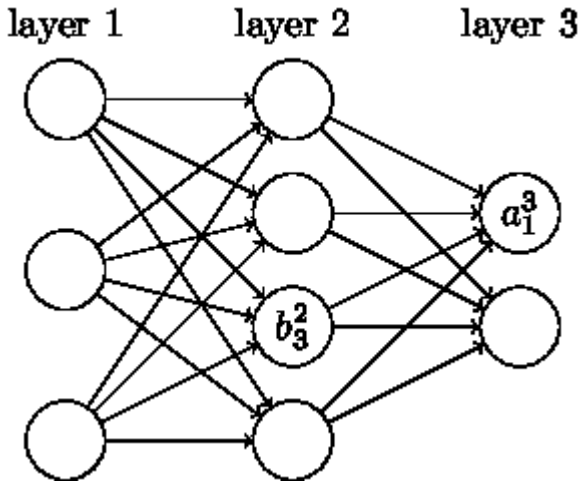
Ogólnie chcemy, by warstwa wewnętrzna miała mniej neuronów niż warstwa wejściowa, aby zmusić sieć do generalizacji — budowania bardziej zwartej reprezentacji danych wejściowych.

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

Przyjmiemy następujące oznaczenia wag połączeń między neuronami: w^l_{jk} 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 *bias*u i aktywacji poszczególnych neuronów: odpowiednio jako b^l_j i a^l_j oznaczymy *bias* oraz aktywację j-tego neuronu w warstwie l-tej. W tej notacji aktywacja j-tego neuronu w warstwie l dana jest wzorem:

$$a^l_j = \sigma(\sum_k w^l_{jk} a^{l-1}_k + b^l_j) \quad \text{w notacji algebry macierzowej:} \quad a^l = \sigma(w^l a^{l-1} + b^l)$$

Uczenie sieci neuronowych — funkcja kosztu

Algorytmy uczenia typowo dobierają wagi i wartości *bias*u 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 $C = \frac{1}{n} \Sigma_x C_x$ 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^l_j obliczeń j-tego neuronu w warstwie l-tej, oraz na tej podstawie pochodne cząstkowe $\partial C / \partial w^l_{jk}$ i $\partial C / \partial b^l_j$. W efekcie można wyznaczyć poprawki dla w^l_{jk} i b^l_j 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 $l = 2, 3, ..., L$ 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 $l = L, L-1, ...3, 2, 1$ 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 $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ i $\frac{\partial C}{\partial b^l_j} = \delta^l_j$ zatem jako poprawki do wag w^l_{jk} i wartości *bias*u b^l_j 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

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 *bias*u 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.

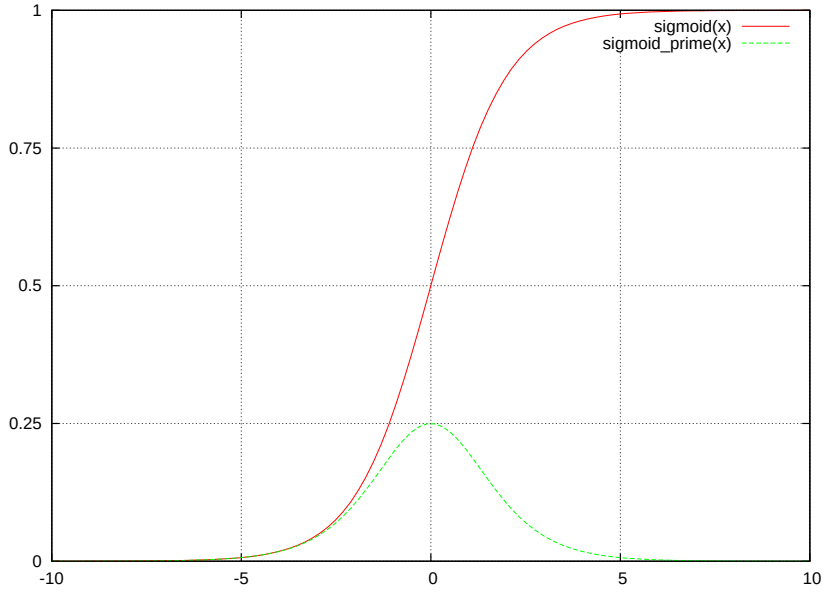
Powyższe pochodne cząstkowe funkcji kosztu można obliczyć jako proporcjonalne do pochodnej funkcji aktywacji $a'(x)$. Zatem wielkość dokonywanych w procesie uczenia poprawek jest związana z pochodną funkcji aktywacji.

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.)

$$
\begin{aligned}
\sigma(x) &= \frac{1}{1+e^{-x}} \\
\sigma'(x) &= \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) \\
\sigma'(x) &= \sigma(x)\cdot(1-\sigma(x))
\end{aligned}
$$

```
$ 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 zobaczyć uruchamiając małe przykłady:
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:

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

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*:

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

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:

$$\sum_j a_j^L = \frac{\Sigma_j \, e^{z_j^L}}{\Sigma_k \, e^{z_k^L}} = 1$$

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 *bias*ach 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.

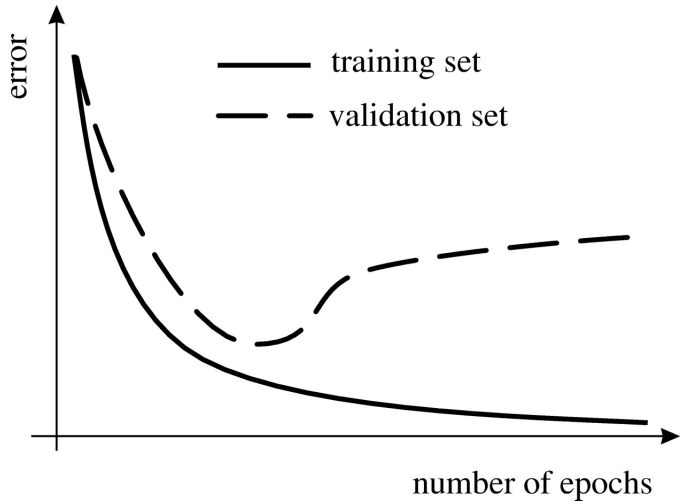
Przeuczenie w sieciach neuronowych

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 pożądany sposób (aby funkcja sieci była bardziej regularna).

[2] Oczywiście, 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.

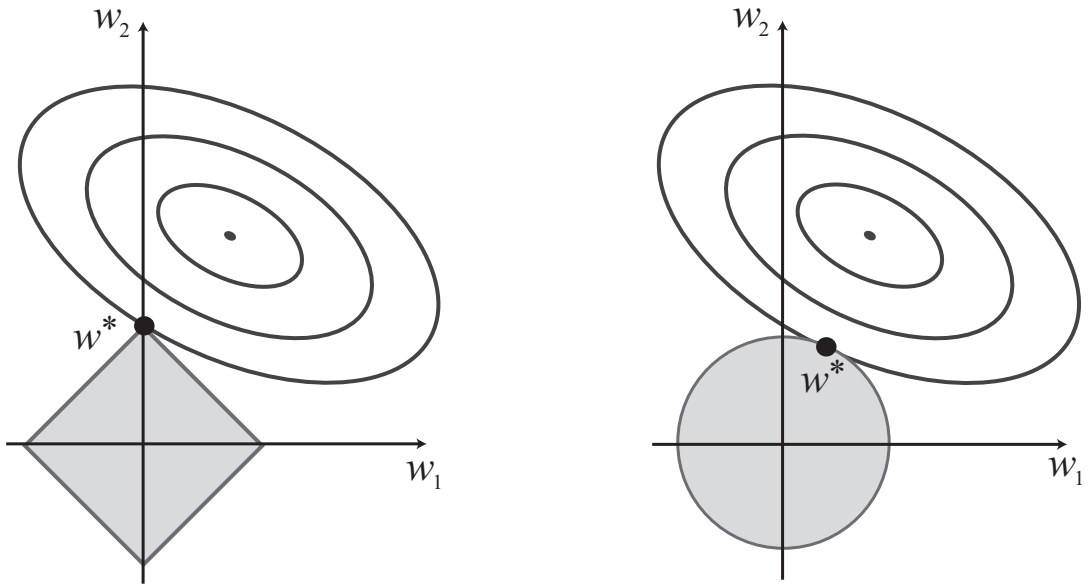
Zauważmy, że czynnik regularyzacji w podanym wyżej wzorze uwzględnia wagi, lecz nie uwzględnia *bias*u. Zatem wartości *bias*u, odmiennie niż wagi, mogą przybierać wielkie wartości. Zwykle nie stanowi to problemu, i z tego powodu nie uwzględnia się *bias*u we wzorze na regularyzację.

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

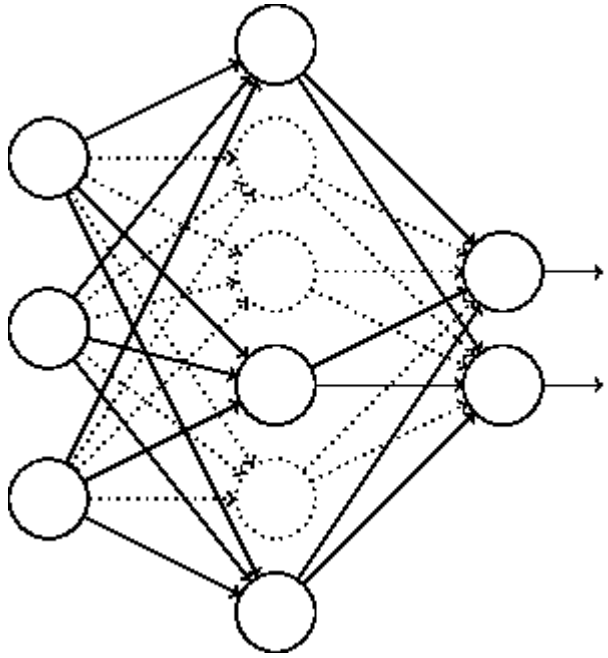
Uwaga terminologiczna: w literaturze maszynowego uczenia oznaczenia L_1 i L_2 stosowane są często zarówno do oznaczenia funkcji kosztu wartości bezwzględnej i średniokwadratowej, jak i wartości bezwzględnej i średniokwadratowej regularyzacji. Kontekst stosowania tych dwóch systemów oznaczeń jest bliski,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

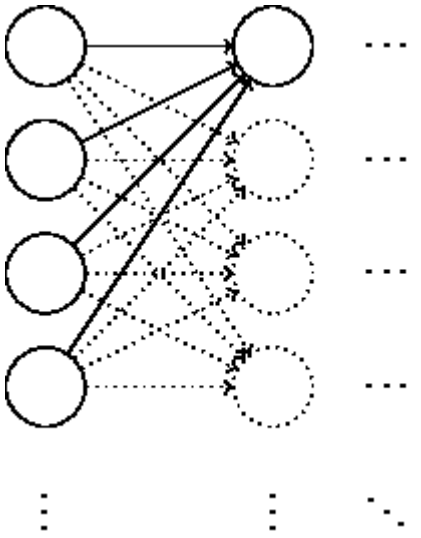
Istnieje szereg interpretacji tego efektu. Podstawowa interpretacja zakłada, że poszczególne neurony muszą „uczyć się" bardziej niezawodnych funkcji aktywacji (wag i *bias*u), ponieważ nie mogą liczyć na wspomaganie ze strony pozostałych neuronów.

Wielokrotne uczenie sieci, jakkolwiek bardziej kosztowne (bo każda faza *dropout*u 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 *bias*ów 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 *bias*ów.



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 *bias*u) $\sqrt{501} \approx 22.4$. 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 *bias*ów nie ma takiego wpływu na szybkość uczenia, i stosuje się albo ich inicjalizację z odchyleniem równym 1, albo nawet inicjalizację *bias*ów wartością zero.

Parametry w uczeniu sieci neuronowych

Głównymi parametrami sieci neuronowej są wagi i *bias*y 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 osiągania 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

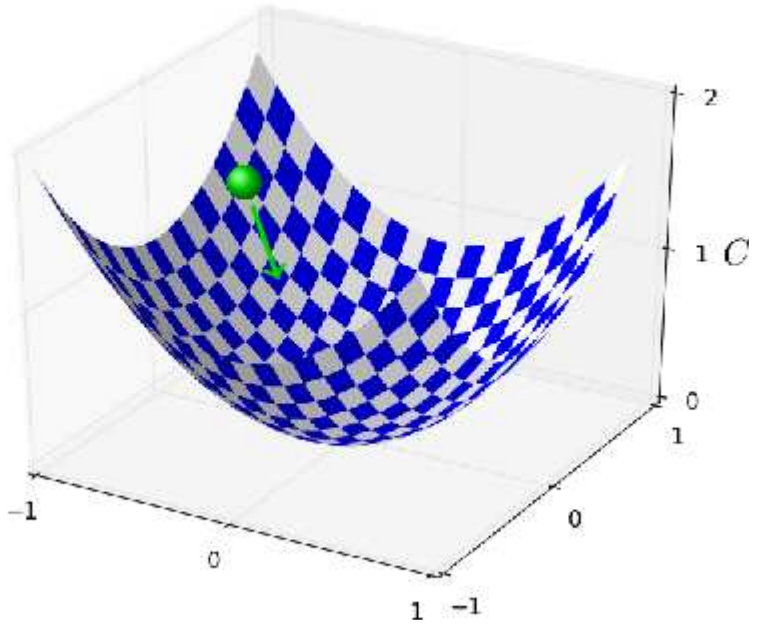
Pierwszym celem w projekcie budowy sieci neuronowej do zadanego celu powinno być osiągnięcie jakiegokolwiek nietrywialnego, czyli istotnie lepszego niż przypadkowy, efektu uczenia. Osiągnąwszy ten wynik, można przystąpić do optymalizacji poszczególnych parametrów. Jednak jego osiągnięcie bywa zadziwiająco 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 pożądanych 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.

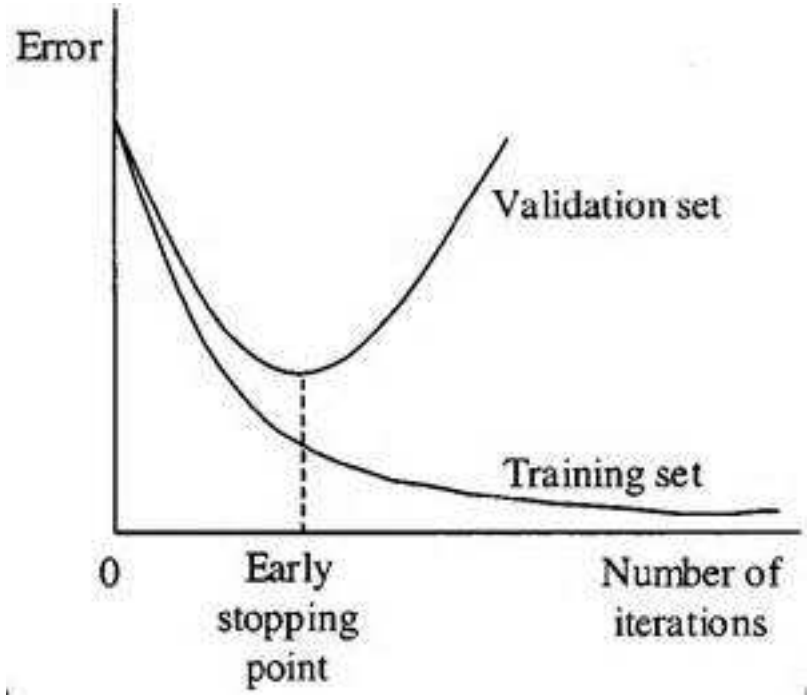


Należy zatem wyznaczyć wartość graniczną η, od której przeszukiwanie od początku poprawia kryterium kosztu. Zaczynając od jakiejś arbitralnej wartości początkowej, takiej jak $\eta = 0.01$, gdy następuje poprawa, można zwiększać η według schematu: $\eta = 0.1, 1.0, ...$ aż do napotkania na oscylacje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

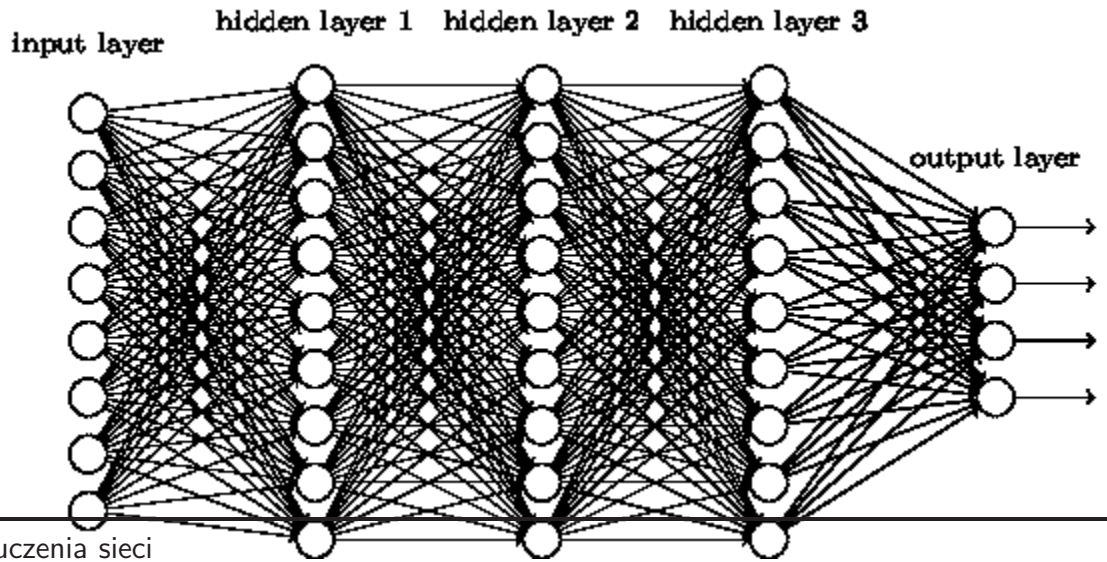
Po określeniu dobrej wartości η można eksperymentować ze współczynnikiem λ zaczynając od arbitralnie wybranej wartości (np. $\lambda = 1.0$) i zwiększać lub zmniejszać tę wartość, oczekując poprawy dokładności na zbiorze walidacyjnym. Po osiągnięciu serii dobrych wartości λ można próbować dostrajać ją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

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

Uczenie sieci głębokich

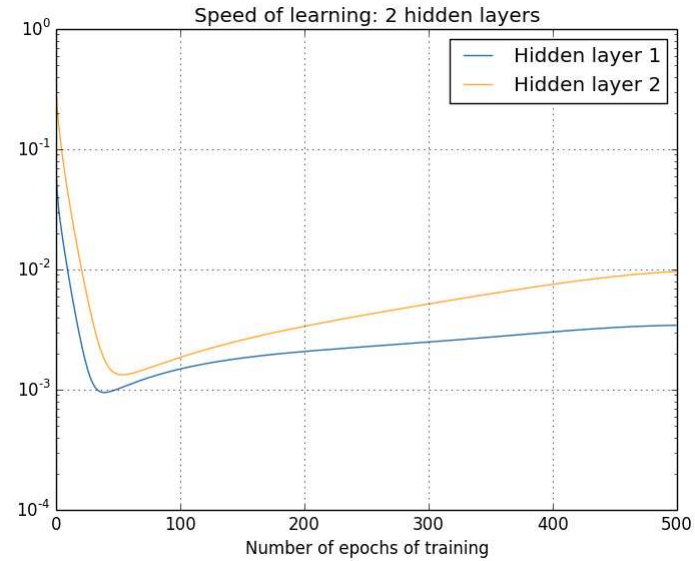
Przez długi okres stosowania jednokierunkowych ANN budowano głównie sieci **płytkie**, typowo z jedną warstwą ukrytą. Wynikało to z faktu, że sieć z jedną lub dwoma warstwami ukrytymi może zamodelować dowolną funkcję. Ponadto, uczenie sieci głębszych jest bardzo powolnym procesem.

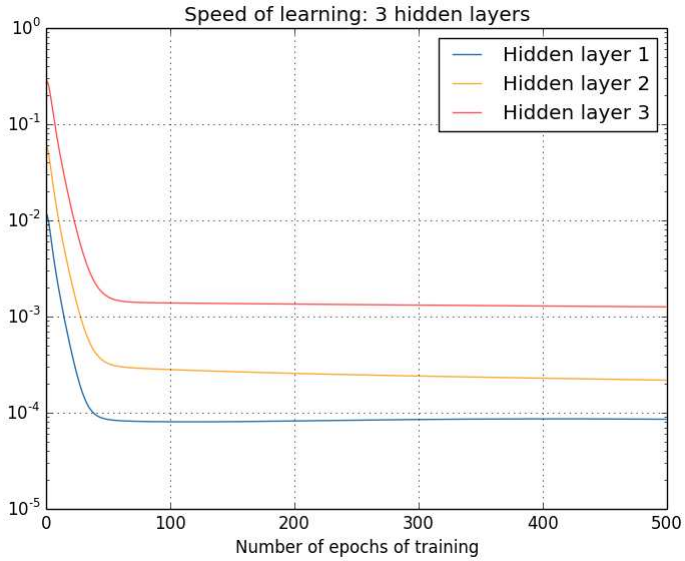
Jednak, intuicyjnie możnaby się spodziewać, że sieć **głęboka**, z wieloma warstwami ukrytymi, może i powinna mieć większe możliwości niż prostsza sieć płytka. Dodatkowe warstwy mogłyby na przykład pozwolić sieci tworzyć kolejne warstwy abstrakcji do analizy problemu, zamiast generować końcowe wyniki w jednym kroku. Ma to szczególne znaczenie w nastającej erze big data, kiedy podejmowane są coraz bardziej złożone zagadnienia, i oczekujemy, że sieci neuronowe będą w stanie zrozumieć i nauczyć się prawidłowości ukrytych w tych danych.

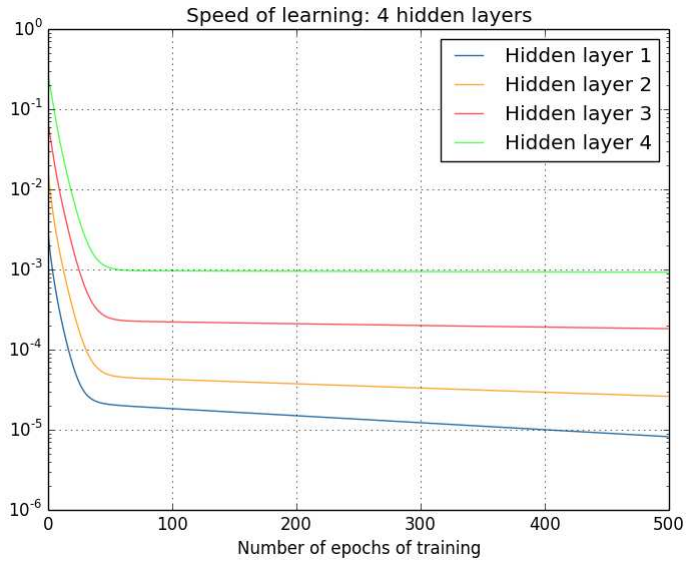


Problem zanikającego gradientu

Próba eksperymentalnej analizy procesu uczenia sieci o większej liczbie warstw ukrytych może wykazać, że w normalnym procesie uczenia metodą propagacji wstecznej gradienty parametrów sieci (wagi i *bias*ów) maleją dramatycznie w kolejno dodawanych początkowych warstwach.





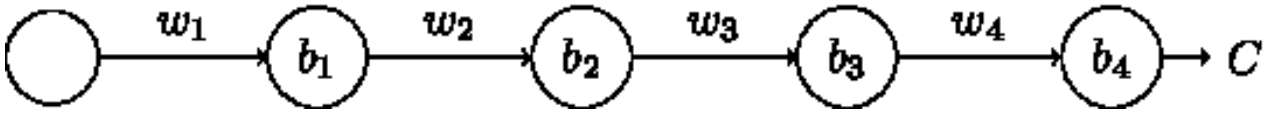


Okazuje się, że jest to zjawisko powszechne. W sieciach głębokich uczenie metodą wstecznej propagacji błędów z funkcją kosztu entropii krzyżowej gradienty kontrolujące szybkość zmian wag, determinujące szybkość uczenia, zanikają dramatycznie.

Problem zanikającego gradientu (cd.)

Pytanie, czy ten **efekt zanikającego gradientu** jest istotnym problemem w uczeniu sieci? Zauważmy, że wagi neuronów zostały początkowo zainicjalizowane wartościami losowymi. Zatem sygnał podawany na wejście sieci jest początkowo całkowicie neutralizowany przez te losowe wagi. Aby sieć zaczęła skutecznie się uczyć, wagi pierwszej warstwy muszą w miarę szybko urosnąć do rozsądnych wartości.
Jednak zmieniają się niezwykle powoli.

Wyjaśnienie problemu zanikającego gradientu można uzyskać analizując trywialnie prostą sieć głęboką, z pojedynczymi neuronami we wszystkich warstwach.



Można wyprowadzić następujący wzór na pochodną cząstkową kosztu po *bias*ie pierwszej warstwy:

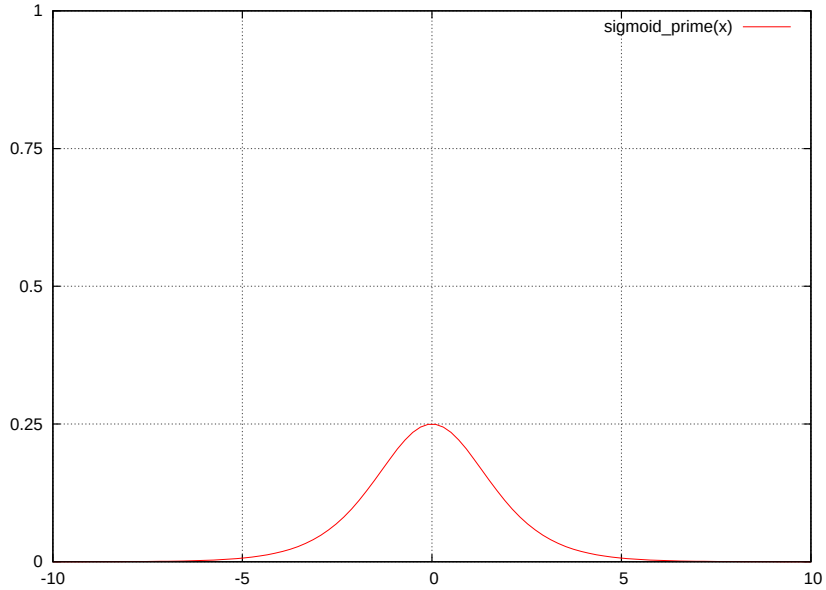
$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, w_2 \sigma'(z_2)\, w_3 \sigma'(z_3)\, w_4 \sigma'(z_4)\, \frac{\partial C}{\partial a_4}$$

Problem zanikającego/eksplodującego gradientu

Aby zbadać zachowanie wyrażenia z poprzedniego slajdu przypomnijmy wzór na pochodną funkcji logistycznej:

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

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



Jak widać, przyjmuje ona dość małe wartości, zwłaszcza dla argumentów istotnie różnych od zera. Mnożąc wiele takich małych czynników, jak we wzorze na poprzednim slajdzie, otrzymujemy bardzo małe wartości gradientu.

Przy dużych wartościach wag — albo samoistnie wyuczonych, albo wymuszonych w algorytmie — jest również możliwe otrzymanie bardzo dużych wartości gradientu, co nazywa się **problemem eksplodującego gradientu**.

W efekcie można wyciągnąć wniosek, że gradient błędu w uczeniu głębokich sieci jednokierunkowych zachowuje się niestabilnie, i nie prowadzi do skutecznego uczenia.

Zatem jak można osiągnąć skuteczne uczenie w sieciach głębokich?

Sieci konwolucyjne

Przedstawiony problem zanikającego (niestabilnego) gradientu parametrów sieci nie jest jedynym problemem utrudniającym budowę i uczenie głębokich sieci neuronowych. Te problemy można rozwiązać budując specjalizowane sieci, których konstrukcja sprzyja skutecznemu uczeniu się.

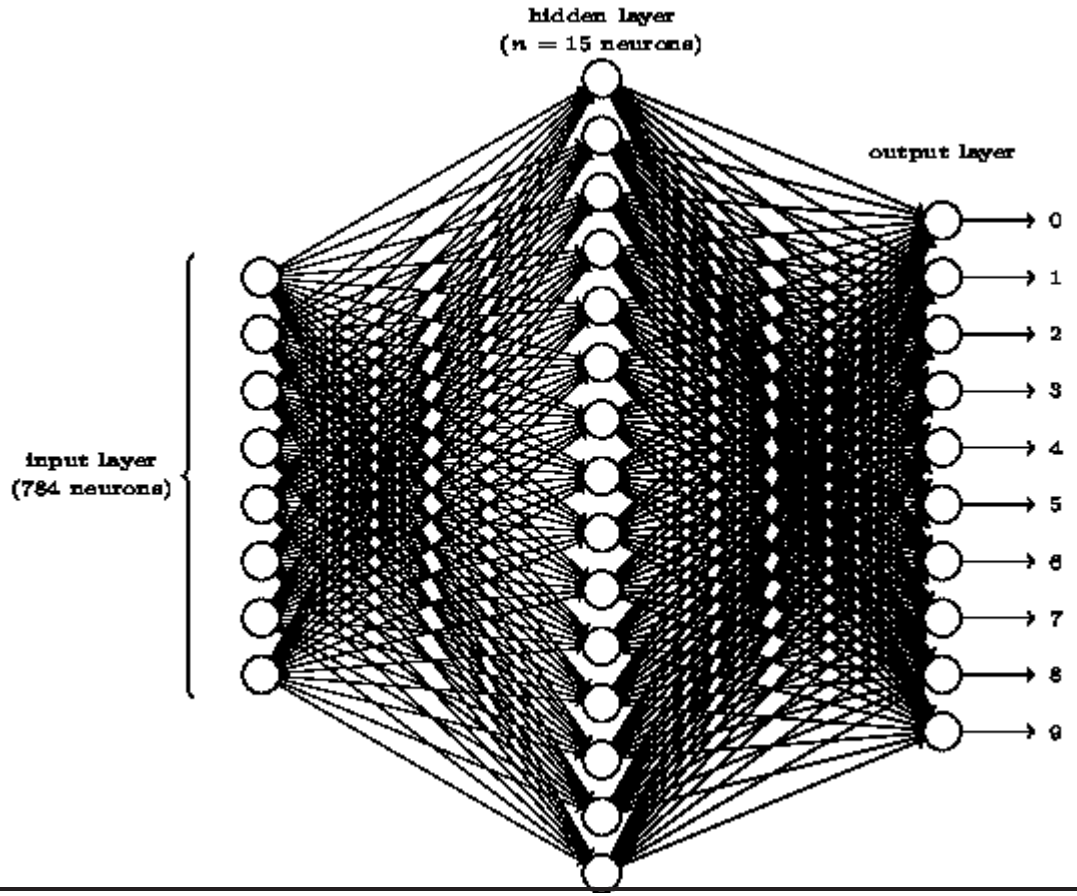
Istnieje szereg modeli sieci głębokich dających dobre efekty uczenia się. Jednym z nich są **sieci konwolucyjne**, typowo nadające się do przetwarzania obrazów. Wynika to z zasady działania operacji **konwolucji** przetwarzającej duży zbiór danych mniejszym filtrem, stopniowo przesuwającym się po zbiorze, i przetwarzającym go fragment po fragmencie. Ten typ przetwarzania znajduje również zastosowanie w innych dziedzinach, jak przetwarzanie sygnału audio, oraz innych danych pomiarowych, a także przetwarzanie wypowiedzi języka naturalnego, itp.

Głęboka sieć konwolucyjna składa się z szeregu warstw o desygnowanych funkcjach, wymuszonych przez specyficzną budowę, wielkość i rodzaj połączeń. W jej budowie wykorzystywane są trzy podstawowe koncepcje: lokalne pola receptywne, wspólne wagi i agregacja obszarów (*pooling*).

Sieci konwolucyjne: dlaczego?

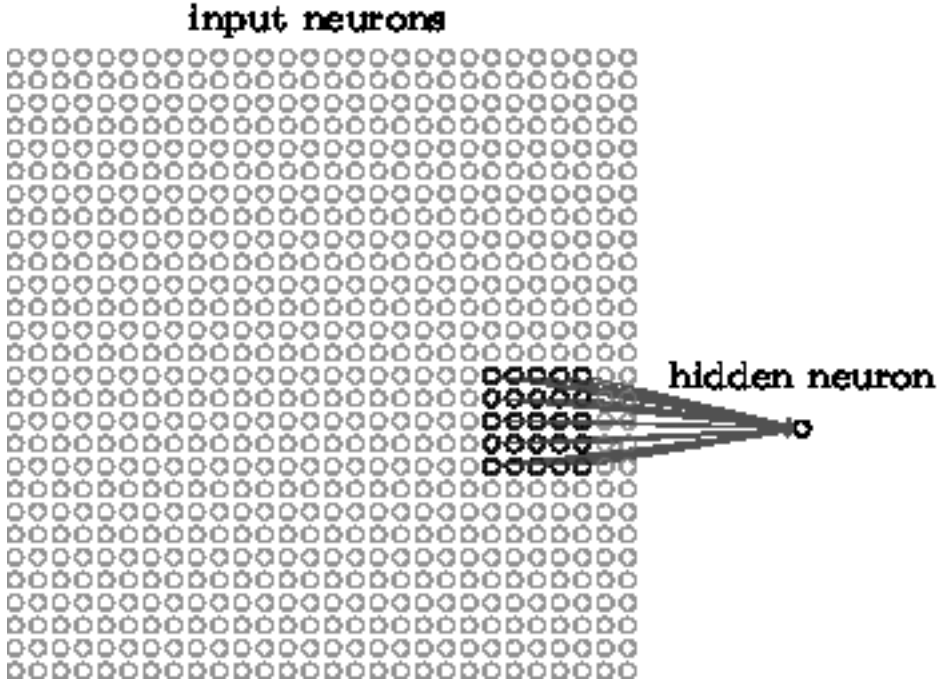
We wspomnianym wcześniej przykładzie analizy obrazów ręcznie pisanych cyfr (baza MNIST) o rozmiarach 28x28 pikseli zastosowana była sieć z pierwszą warstwą (wejściową) o rozmiarze 784 (=28x28) pikseli, z kompletem połączeń do wszystkich neuronów warstwy drugiej (ukrytej). Ma to sens z punktu widzenia przetwarzania kompletnych instancji danych wejściowych. Jednak z punktu widzenia przetwarzania obrazu jest to dziwne, ponieważ druga warstwa ma za zadanie nauczyć się przetwarzania zarówno pikseli sąsiadujących, jak i położonych daleko od siebie.

Warstwa wejściowa jest widziana jako jednowymiarowy rząd neuronów, nieodzwierciedlający ich sąsiedztwa.



Warstwa konwolucji

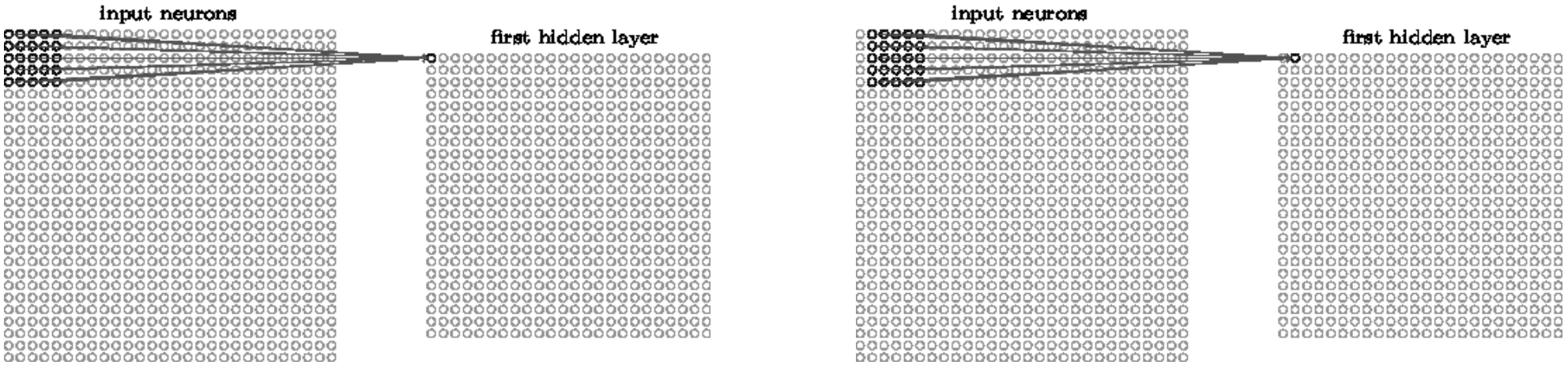
Jak w takim razie zorganizować przetwarzanie sygnałów z warstwy wejściowej? Pod uwagę brane są małe obszary obrazu traktowane jako **lokalne pola receptywne** (*local receptive fields*).



Drugą warstwę sieci stanowić będzie warstwa konwolucji, przetwarzająca każde kolejne lokalne pole receptywne filtrem konwolucyjnym. Filtr jest niewielką tablicą (macierzą) przetwarzającą wszystkie neurony pola receptywnego, odwzorowującą je na pojedynczy neuron drugiej warstwy. Na przykład, macierz konwolucji może mieć rozmiar 5x5.
Zamiast pełnego zestawu połączeń z pierwszej do drugiej warstwy, będziemy mieli tylko połączenia lokalnych pól receptywnych z pojedynczymi neuronami warstwy konwolucji.

Przetwarzanie konwolucyjne

Przetwarzanie obrazu filtrem konwolucyjnym polega na zastosowaniu filtra do pierwszego pola receptywnego, obliczeniu wartości wyjściowej, i przesunięciu filtra na kolejne pole receptywne.



Wartość tego przesunięcia, albo kroku konwolucji (*stride*), na powyższym rysunku wynosi 1 neuron.

Arytmetyka konwolucji

Rozważmy konkretny przykład: sieć przetwarzającą obraz o rozmiarze 28x28 pikseli dla przetwarzania ręcznie zapisanych cyfr bazy MNIST.

Załóżmy, że zastosujemy macierz konwolucji o rozmiarze 5x5. Przyjmując, że macierz będzie przesuwana po jednym pikselu w każdym kierunku (poziomym i pionowym), istnieją 24 położenia macierzy konwolucji w obu wymiarach, a zatem metoda takiego przesuwania macierzy konwolucji da na wyjściu dane dla 24x24 neuronów, i stąd taki musi być rozmiar warstwy konwolucji (pierwszej warstwy ukrytej).

W ten sposób zaledwie 25 wag i jedna wartość *bias*u obsługuje połączenie pomiędzy pierwszą i drugą warstwą sieci. Jest to zasadnicza różnica pomiędzy taką architekturą a kompletem połączeń pomiędzy tymi warstwami, które wymagałby prawie pół miliona wag (28x28x24x24) i 576 *bias*ów. Dzięki temu uczenie sieci może być efektywne.

Zauważmy jeszcze, że druga warstwa mogłaby teoretycznie mieć dokładnie ten sam rozmiar co przetwarzany obraz, co wymagałoby przykładania środka macierzy konwolucji do samej granicy obrazu, częściowo poza jego granicą, i zasilanie jej wartościami nieistniejących pikseli. Warstwa konwolucji mogłaby również być istotnie mniejsza od warstwy wejściowej przez zastosowanie przesunięcia (*stride*) o dwa neurony (lub więcej), co wygenerowałoby warstwę konwolucji o rozmiarze 12x12.

Współdzielenie wag warstwy konwolucyjnej

Filtrowanie za pomocą konwolucji jest bardzo dobrze znane i popularne w przetwarzaniu obrazów. Stosowane są różne typy filtrów konwolucyjnych (inaczej: splotowych): rozmycie, wyostrzanie, wykrywanie krawędzi, i inne.

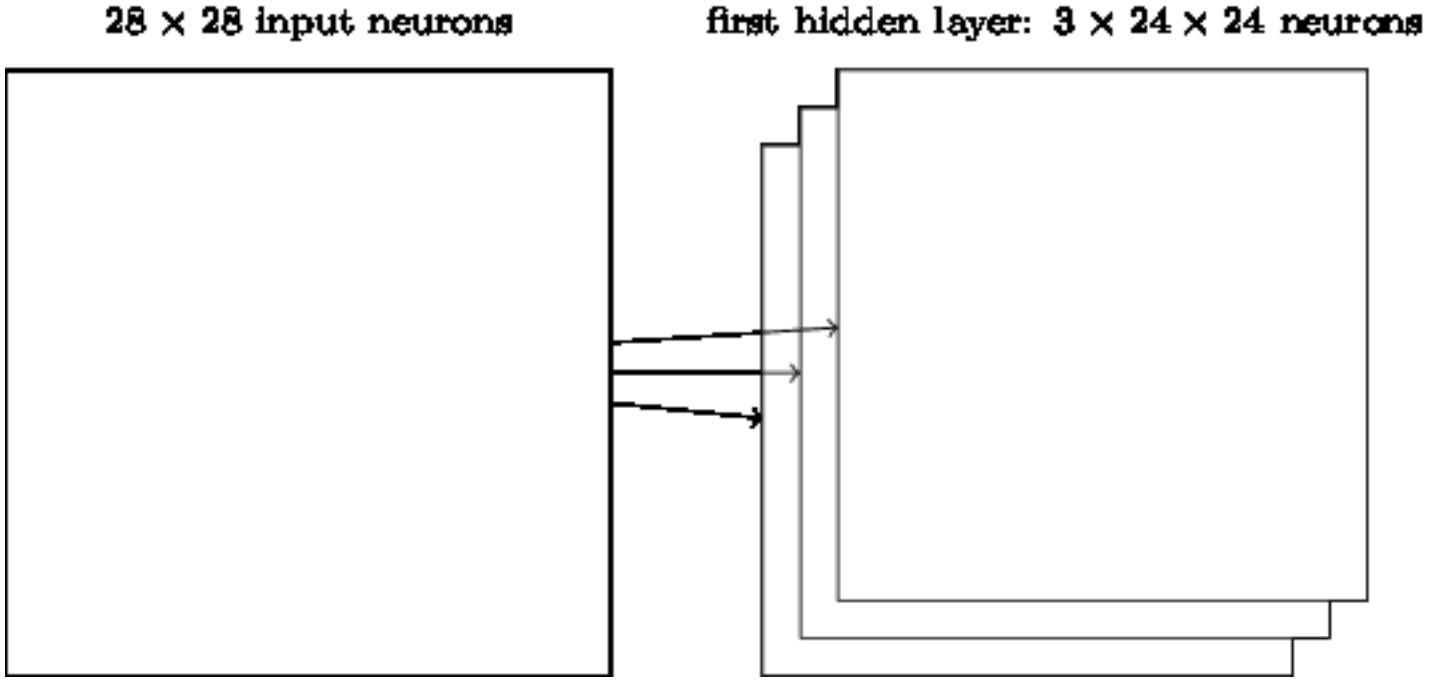
Więc który filtr splotowy będzie używany w naszej konkretnej sieci? Kluczową ideą stosowaną w głębokich sieciach konwolucyjnych jest **współdzielenie wag**. Wszystkie filtry splotowe przetwarzające kolejne lokalne pola receptywne wykorzystują ten sam zestaw 25 wag (plus bias). Przyspiesza to proces uczenia i zmusza wagi do dostrojenia się do przetwarzania identycznego dla wszystkich lokalnych pól receptywnych.

W efekcie warstwa konwolucji, którą właśnie utworzyliśmy, nauczy się rozpoznawać pewną cechę w obrazie. Ta cecha może mieć prostą interpretację, taką jak centralna plama lub krawędź biegnąca pod określonym kątem, ale może też być jakimś trudnym do opisania wzorcem (choć musi być prosty, ze względu na rozmiar filtra). Prostokątna struktura neuronów wygenerowana przez filtr konwolucji nazywana jest **mapą cech**.

Ważne jest zrozumienie, że cecha, którą nasza sieć uczy się rozpoznawać, będzie losowa — wynikająca z przypadkowej inicjalizacji wag i biasu filtra. Ale zarazem, ta wyuczona cecha będzie rozpoznawana niezależnie od jej położenia na całym obrazie.

Wiele filtrów konwolucyjnych

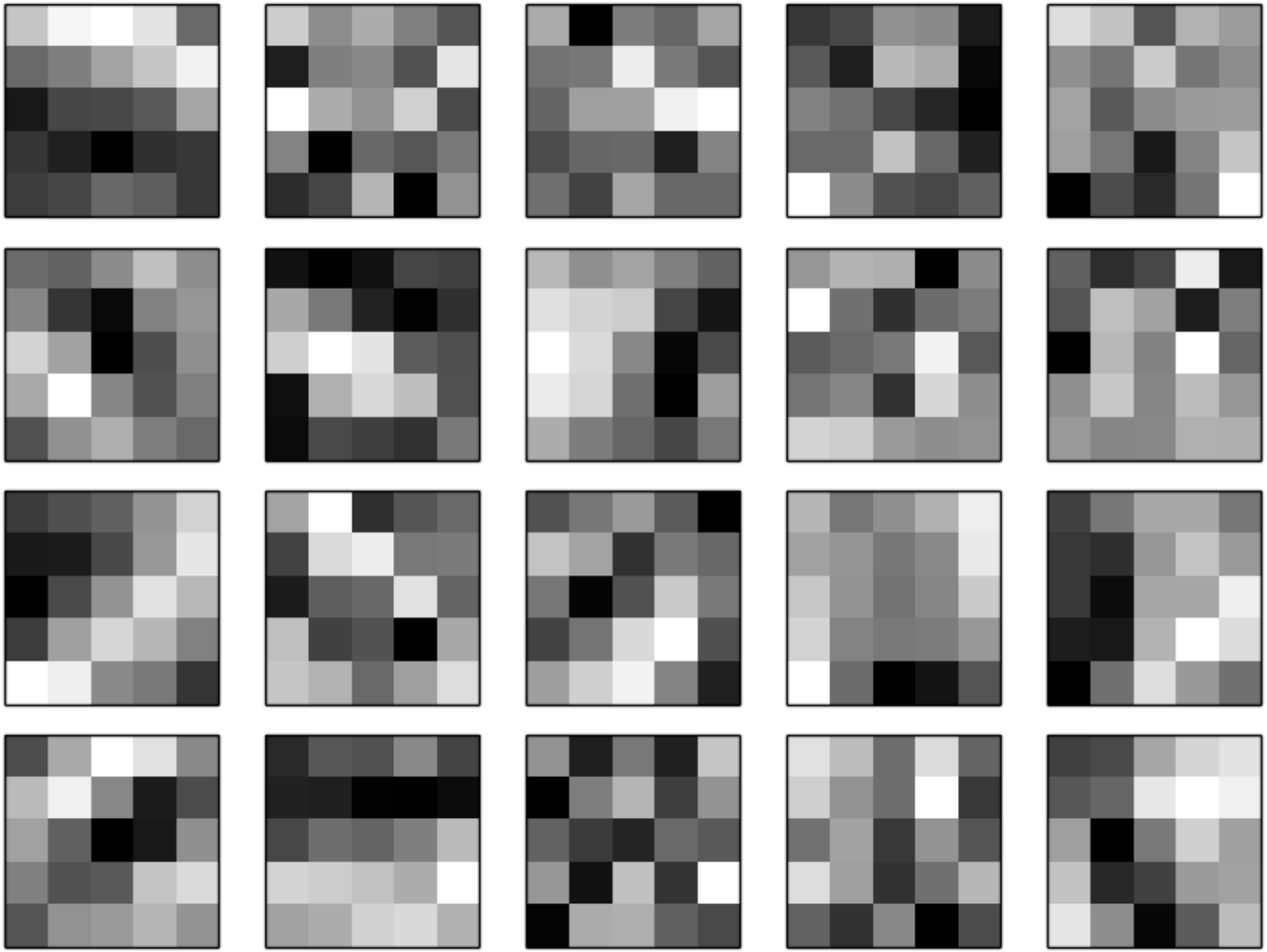
Warstwa splotu uczy się rozpoznawać losową cechę obrazu. Ta cecha może być mniej lub bardziej przydatna w interpretacji całego obrazu. Możemy zmaksymalizować szansę na uzyskanie przydatnych filtrów wykrywających cechy, tworząc ich kilka lub nawet wiele. Dzięki randomizacji, można oczekiwać, że każdy będzie inny.



Na powyższym obrazku mamy trzy mapy cech. Są one równolegle i niezależnie wytrenowane, zatem wspólnie razem tworzą drugą warstwę (konwolucyjną) naszej sieci, która jest zarazem pierwszą warstwą ukrytą.

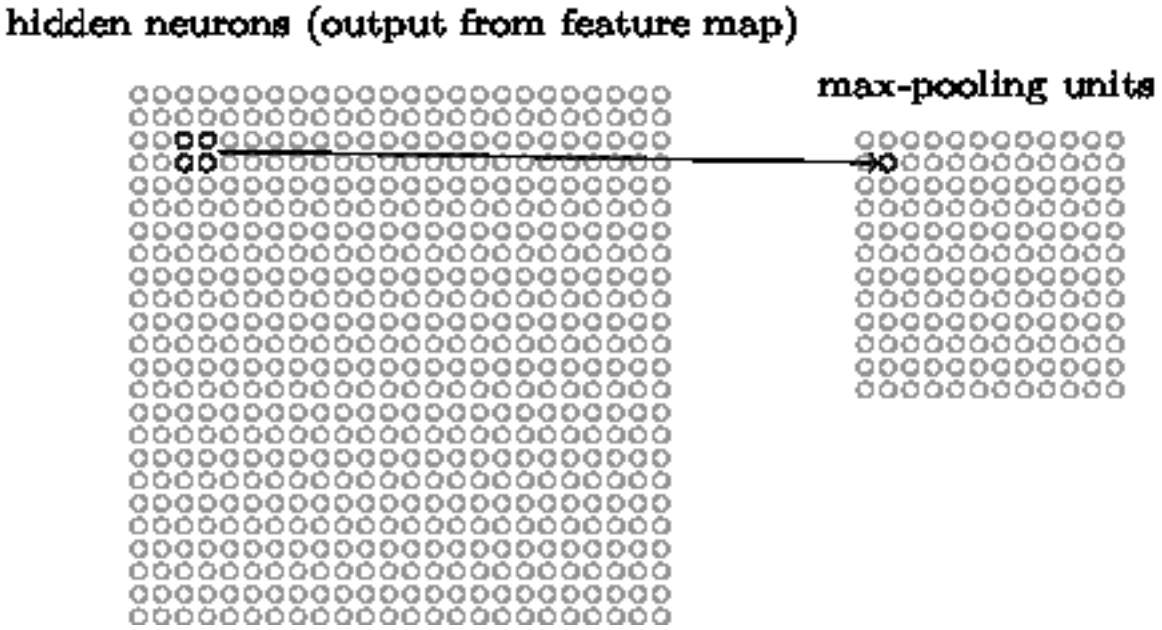
Wiele filtrów konwolucyjnych (cd.)

Przedstawione powyżej trzy mapy cech to tylko przykład. Można stworzyć znacznie więcej. Oto przykładowe filtry wytrenowane na bazie danych MNIST:



Warstwy agregacji

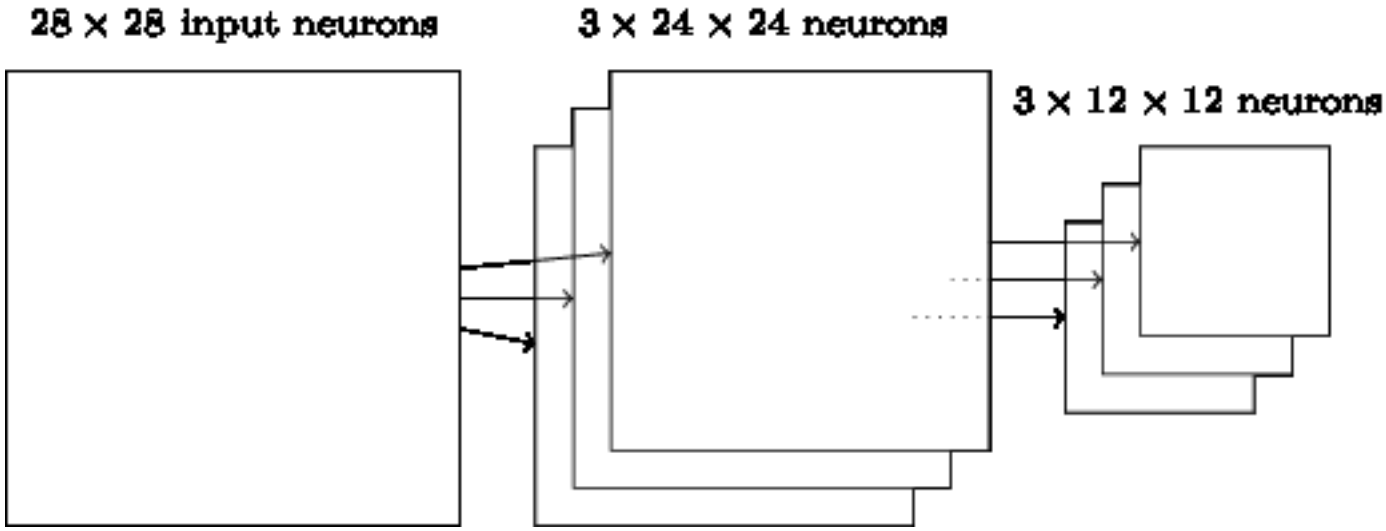
Mapy cech utworzone w warstwie splotu reprezentują rozpoznane drobne cechy oryginalnego obrazu. Każda z nich jest jednak prawie tak duża, jak obraz oryginalny (zależy to od wielkości kroku, plus marginesów wynikających z wielkości filtra splotu). Można powiedzieć, że mapy cech są tworzone w pełnej rozdzielczości oryginalnego obrazu. Ale w rzeczywistości poszukujemy na obrazie cech makroskopowych. Potrzebujemy więc sposobu **zagregowania** obrazu, które zachowa rozpoznane cechy.



W tym celu można zbudować kolejną warstwę sieciową realizującą **agregację** (*pooling*). Jedną z powszechnie używanych funkcji agregacji, jest max, obliczająca maksymalną wartość piksela na pewnym obszarze dużej mapy obiektów.

Warstwy agregacji (cd.)

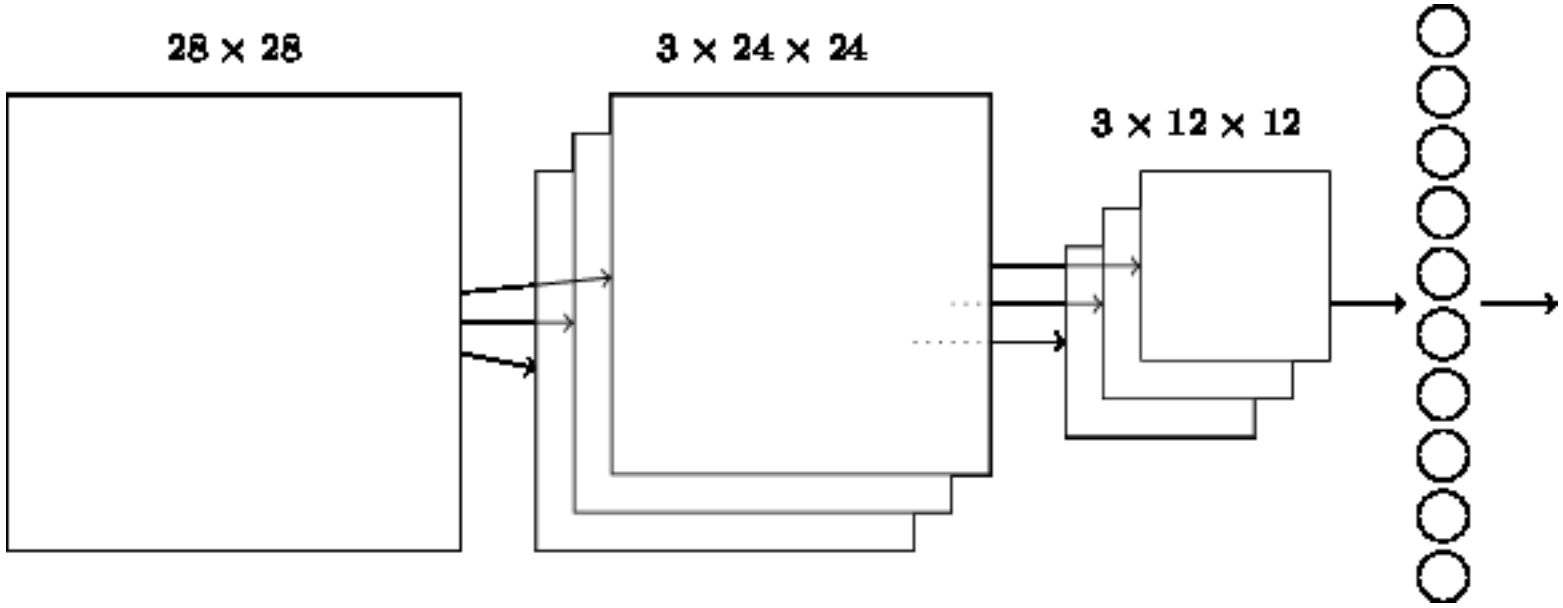
W prezentowanym przykładzie zastosowano agregacji 2x2 obszarów neuronowych. Odpowiada to zmniejszeniu rozmiaru piksela obrazu o połowę, z 24x24 do 12x12. Każda mapa obiektów z warstwy konwolucji powinna być agregowana w ten sam sposób, co daje następującą strukturę:



Zamiast agregacji funkcją max można było użyć innej funkcji agregacji. Przykładem jednej z nich jest **agregacja L2**, które oblicza pierwiastek kwadratowy z sumy kwadratów aktywacji neuronów w obszarze podlegającym agregacji. Optymalny wariant można wypracować, eksperymentując z różnymi możliwościami.

Uzupełnienie sieci

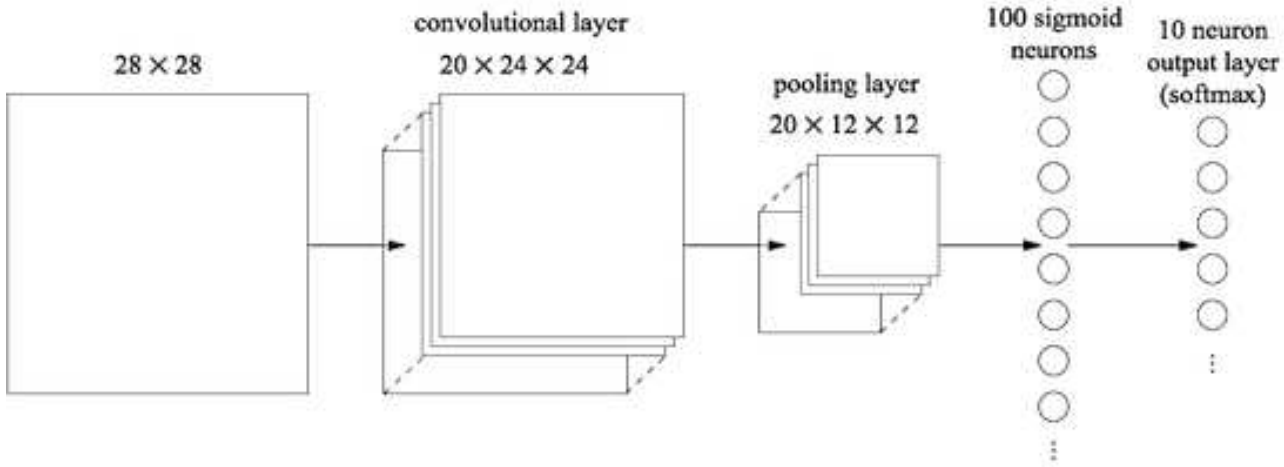
Jeśli uznamy, że utworzone do tej pory warstwy sieciowe powinny wystarczyć do rozpoznania odręcznych cyfr, możemy zakończyć dodając warstwę wyjściową składającą się z dziesięciu neuronów, z których każdy wytrenowany jest do odpowiedzi na inną cyfrę, tak jak poprzednio:



Oczekujemy, że neurony warstwy wyjściowej rozpoznają cyfrę zawartą w obrazie wejściowym poprzez analizę cech zidentyfikowanych przez sieć. Z tego powodu każdy neuron wyjściowy musi mieć połączenia ze wszystkich neuronów poprzedniej warstwy (agregacji). Innymi słowy, między dwiema ostatnimi warstwami musi zostać zainicjalizowane pełne połączenie.

Dalsze szczegóły

Przy budowaniu kompletnej sieci dla rozpoznawania ręcznie pisanych cyfr z bazy MNIST, ma sens zastosowanie w ostatniej warstwie neuronów softmax z jednoczesnym zastosowaniem funkcji kosztu wiarygodności logarytmicznej (*log-likelihood*). Przy użyciu dwudziestu równoległych filtrów w warstwie konwolucji sieć ma następującą postać:



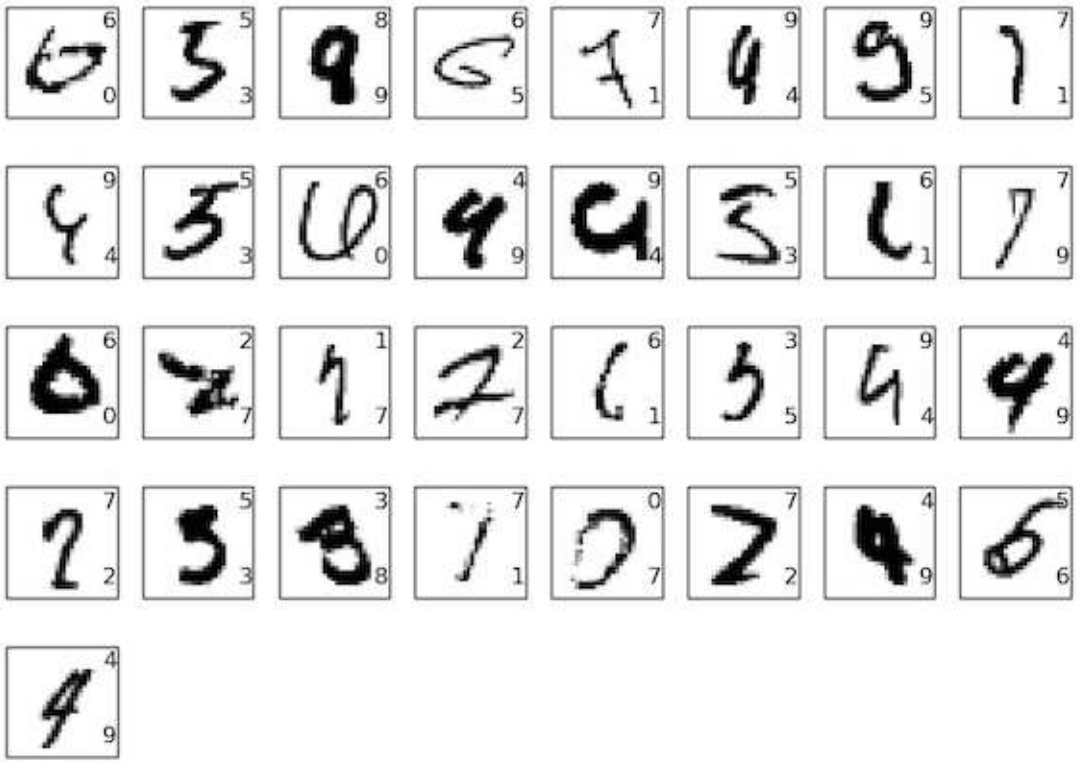
Dodatkowa, w pełni połączona warstwa neuronów z sigmoidalną funkcją aktywacji ma na celu zintegrować informacje generowane przez indywidualne filtry konwolucyjne, umożliwiając końcowej warstwie neuronów softmax podjęcie finalnej decyzji.

Inne warianty

Klasyfikacja ręcznie zapisanych cyfr bazy MNIST była intensywnie badanym problemem i powstało wiele klasyfikatorów i prac naukowych demonstrujących skuteczne techniki pozwalające ulepszyć otrzymane wyniki. Należą do nich:

- Zastosowanie drugiego kompletu warstw konwolucyjnej i grupującej, analizującej wyniki z pierwszej takiej warstwy. Ma to efekt wprowadzenia dodatkowego poziomu abstrakcji, mającego wykrywać cechy obrazu jeszcze bardziej makroskopowe. Jednak druga warstwa konwolucyjna ma na wejściu połączone wyjścia wszystkich wyjść pierwszej warstwy grupującej.
- Użycie większego zbioru danych. Jakkolwiek często dodatkowych danych nie ma, jak w przypadku bazy MNIST, jednak możliwe okazuje się „rozmnażanie" danych, przez transformacje istniejących obrazów takie jak: rotacje i przesunięcia. Poza większą skutecznością uczenia, takie rozszerzenie zbioru danych zmniejsza również skłonność do przeuczenia.
- Użycie metody *ensemble learning*, to znaczy wytrenowanie kilku niezależnych sieci, i dodanie warstwy głosowania wybierającej wynik metodą większościową. Zastosowanie tej metody opiera się na założeniu, że poszczególne sieci mogą popełniać różne błędy, i takie błędy mogą być wyeliminowane.
- Wykorzystanie innych funkcji aktywacji, takich jak tanh, albo szczególnie relu.

33 przypadki błędnej klasyfikacji na 10000 obrazach bazy MNIST (górny indeks wskazuje rzeczywiście zapisaną cyfrę, a dolny indeks cyfrę rozpoznaną przez sieć głęboką):



Uczenie sieci konwolucyjnych

Uczenie sieci konwolucyjnej może być realizowane podobnie jak uczenie wielowarstwowego perceptronu z pełnym zestawem połączeń pomiędzy kolejnymi warstwami. To znaczy możliwe jest wykorzystanie propagacji wstecznej błędów, i optymalizacji wykorzystującej metodę stochastycznego największego gradientu.

Procedura propagacji wstecznej musi być dostosowana do specyfiki sieci konwolucyjnej, ale nie stanowi to problemu. Co więcej, ze względu na niezależność elementów sieci konwolucyjnej, uczenie może przebiegać w sposób równoległy, i poza wykorzystaniem jednostek wieloprocesorowych i/lub wielu rdzeni, możliwe jest wykorzystanie wielu procesorów na karcie graficznej GPU.

Uczenie transferowe

Uczenie transferowe (*transfer learning*) to grupa technik zmierzająca do tego by wykorzystać wiedzę zdobytą w trakcie maszynowego uczenia się jednego zadania, lub gotowe modele stworzone w jego trakcie, do innego zadania maszynowego uczenia.

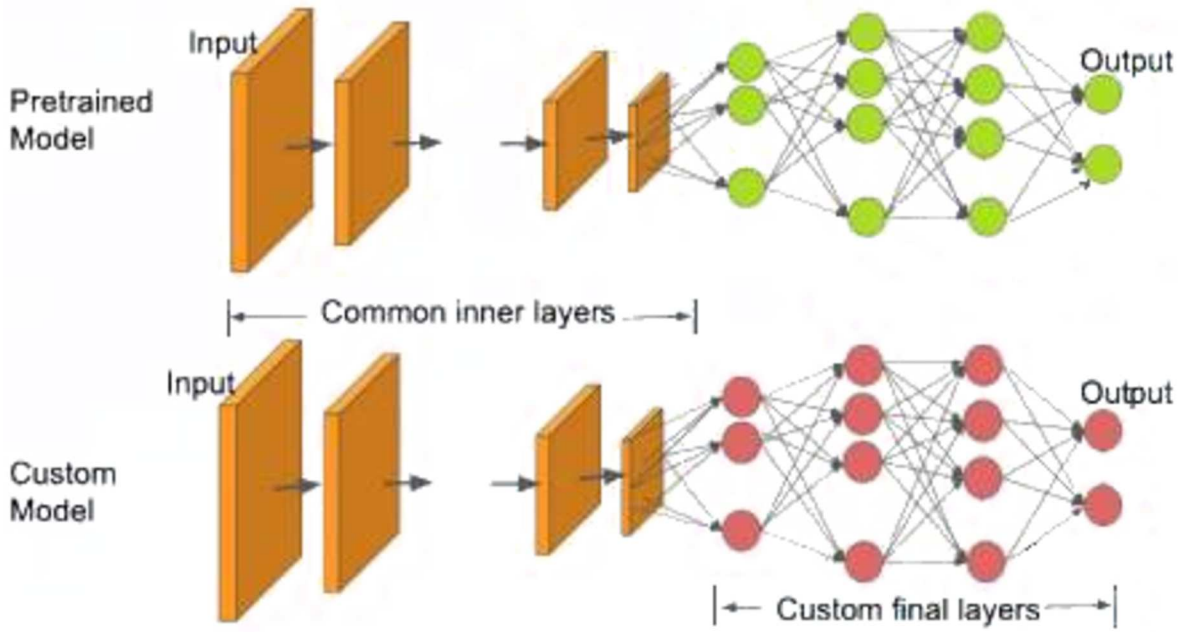
To podejście, mające pokrewne mechanizmy w psychologii, i rozważane w badaniach nad uczeniem maszynowym od lat 1970-tych. Obecnie jest coraz częściej stosowane w odniesieniu do sieci głębokich jako alternatywa do budowania każdej aplikacji, czyli uczenia sieci neuronowej, od podstaw.

Uczenie transferowe jest często stosowane w przypadkach:

- aby zaoszczędzić czas i zasoby wynikające z konieczności trenowania od podstaw wielu modeli uczenia maszynowego w celu wykonania podobnych zadań,
- jako oszczędność wydajności w obszarach uczenia maszynowego, które wymagają dużych zasobów, takich jak kategoryzacja obrazów lub przetwarzanie języka naturalnego,
- aby zniwelować brak etykietowanych danych treningowych w posiadaniu organizacji poprzez wykorzystanie wstępnie wytrenowanych modeli.

Uczenie transferowe na sieciach konwolucyjnych

Przykładem uczenia transferowego może być rozpoznawanie obrazów. W sieci głębokiej wytrenowanej na dużym zbiorze uczącym zwykle początkowe warstwy sieci rozpoznają abstrakcyjne cechy graficzne takie jak punkty i krawędzie, dalsze warstwy rozpoznają bardziej złożone elementy, określone kształty będące elementami różnych obrazów, i bardziej złożone struktury, przydatne do rozpoznawania konkretnych obiektów.



Zwróćmy uwagę, że aby możliwe było "douczenie" zaadaptowanego modelu głębokiego, rozmiar próbek uczenia docelowego musi być zgodny z architekturą początkowych warstw sieci (niepodlegających uczeniu). Gdyby ten rozmiar był niezgodny, to uczenie transferowe wymagałoby wstępnej konwersji wszystkich próbek danych.

Uczenie transferowe na sieciach konwolucyjnych (cd.)

Jakkolwiek ostatnia warstwa jest precyzyjnie dostrojona do rozpoznawania obiektów do których została zbudowana sieć, to w trakcie uczenia transferowego można poddać uczeniu tylko tę warstwę, pozostawiając początkowe warstwy niezmienione. Dlatego w trakcie uczenia z wykorzystaniem propagacji wstecznej błędów należy **zamrozić** (*freeze*) wyuczone wagi wszystkich warstw, które mają zostać wykorzystane z gotowego modelu. Pozwala to wykorzystać zgromadzoną w nich wiedzę pozyskaną z dużego zbioru uczącego, aby zbudować sieć za pomocą mniejszego zbioru innych obrazów.

Przykłady konwolucyjnych modeli głębokich dobrze wytrenowanych do rozpoznawania obrazów i dostępnych w necie:

- VGG-16
- VGG-19
- Inception V3
- XCeption
- ResNet-50

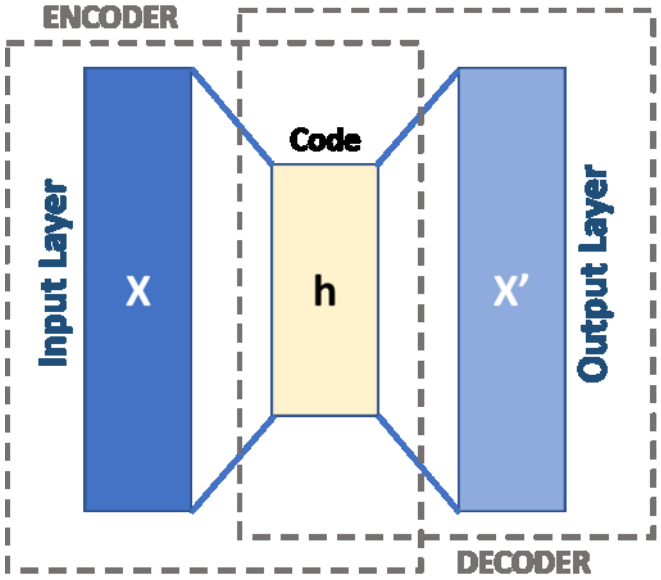
Istnieją również wytrenowane modele głębokie wykorzystywane do przetwarzania języka naturalnego.

Autoenkodery

Autoenkoder to sieć neuronowa, której celem jest nauczenie się reprezentacji zestawu danych, przez ich zakodowanie do bardziej zwartej postaci alternatywnej. Jedną z jego realizacji jest architektura sieci podobna do jednowarstwowego perceptronu, w którym na wyjściu pojawia się sygnał identyczny do wejściowego. Sieć musi nauczyć się wiernego odtwarzania sygnału wejściowego, ale poprzez skróconą postać wewnętrzną.

Autoenkoder składa się z dwóch części:

- kodera, który przekształca dane wejściowe do reprezentacji wewnętrznej,
- dekodera, który przekształca dane z reprezentacji wewnętrznej, do oryginalnych danych wejściowych.



Autoenkodery — warianty

Zauważmy, że podstawowa architektura autoenkodera jest formą uczenia nienadzorowanego, ponieważ sieć uczy się na danych nieetykietowanych. Dowolny zbiór danych może być zbiorem uczącym, jak w uczeniu nienadzorowanym.

Autoenkodery mogą być stosowane jako algorytm redukcji wymiaru, ponieważ w przypadku warstwy ukrytej o rozmiarze mniejszym od rozmiaru wejścia, autoenkoder jest zmuszony do nauczenia się bardziej zwartej reprezentacji danych.

Poza podstawową architekturą, w której warstwa wewnętrzna ma rozmiar mniejszy od wejścia, można rozważać autoenkodery z warstwą wewnętrzną o rozmiarze równym, albo nawet większym od rozmiaru wejścia.

W takich układach potrzebne jest zastosowanie dodatkowych technik aby nie dopuścić do nauczenia się prostego odtwarzania sygnały wejściowego na wyjściu. Te techniki mają formę różnych funkcji kary i są podobne do technik regularyzacji poznanych wcześniej.

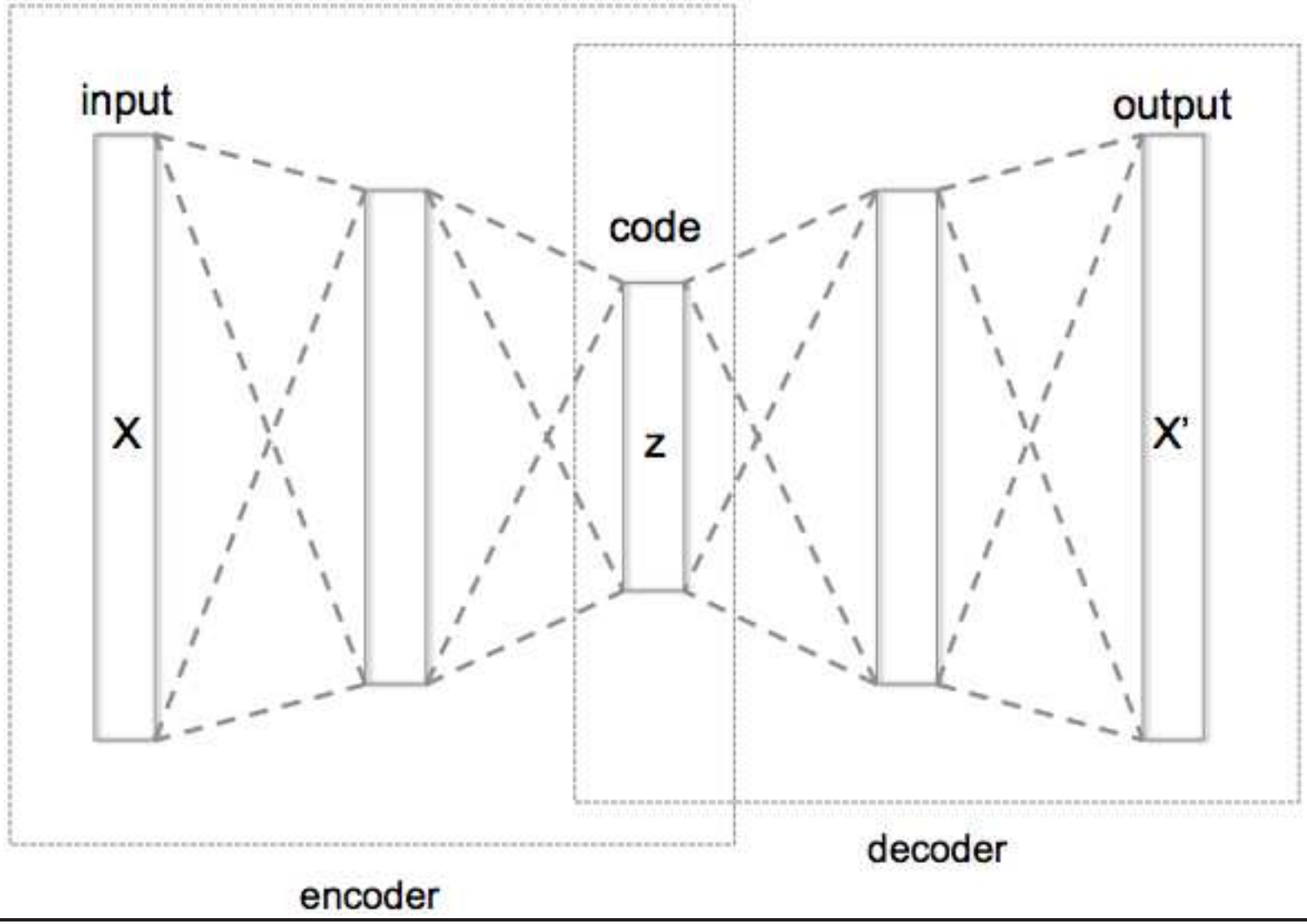
Autoenkodery — zastosowania

Zastosowania autoenkoderów:

- redukcja wymiaru — najbardziej bezpośrednie zastosowanie autoenkoderów, będące oryginalnym celem i motywacją stworzenia ich koncepcji
- uzyskiwanie informacji (*information retrieval*) — staje się łatwiejsze i bardziej skuteczne dzięki redukcji wymiaru
- wykrywanie anomalii — autoenkoder wytrenowany na danych „normalnych" wykazuje zwiększone błędy rekonstrukcji na danych „anomalnych", co pozwala je wykrywać
- kompresja obrazów — algorytmy oparte na autoekoderach są niekiedy lepsze od innych współczesnych metod stratnej kompresji obrazów
- odszumianie obrazów — autoenkoder jest uczony odtwarzać poprawną postać sygnału przez uczenie z zaszumionymi sygnałami na wejściu
- maszynowe tłumaczenie tekstów — sekwencje słów na wejściu są kodowane do reprezentacji wewnętrznej, natomiast dekoder tworzy na wyjściu tekst w języku docelowym

Autoenkodery głębokie

Poza podstawową architekturą autoenkodera z jedną warstwą wewnętrzną, można tworzyć autoenkodery głębokie z wieloma warstwami wewnętrznymi. Autoenkodery głębokie pozwalają uzyskać lepszą kompresję, a także skutecznie uczyć się na mniejszych zbiorach danych.



Materiały

W tej prezentacji wykorzystane zostały głównie materiały z następujących prac:

1. Michael A. Nielsen: Neural Networks and Deep Learning, Determination Press, 2015
`http://neuralnetworksanddeeplearning.com/`

2. Ian Goodfellow, Yoshua Bengio, Aaron Courville: Deep Learning, MIT Press, 2016
`http://www.deeplearningbook.org`