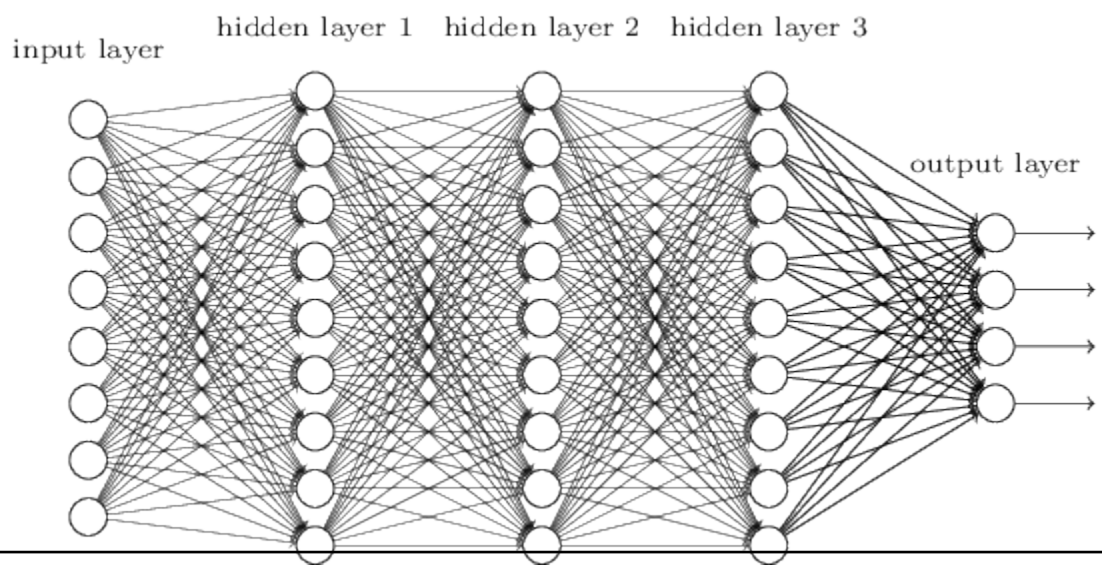
Uczenie sieci głębokich

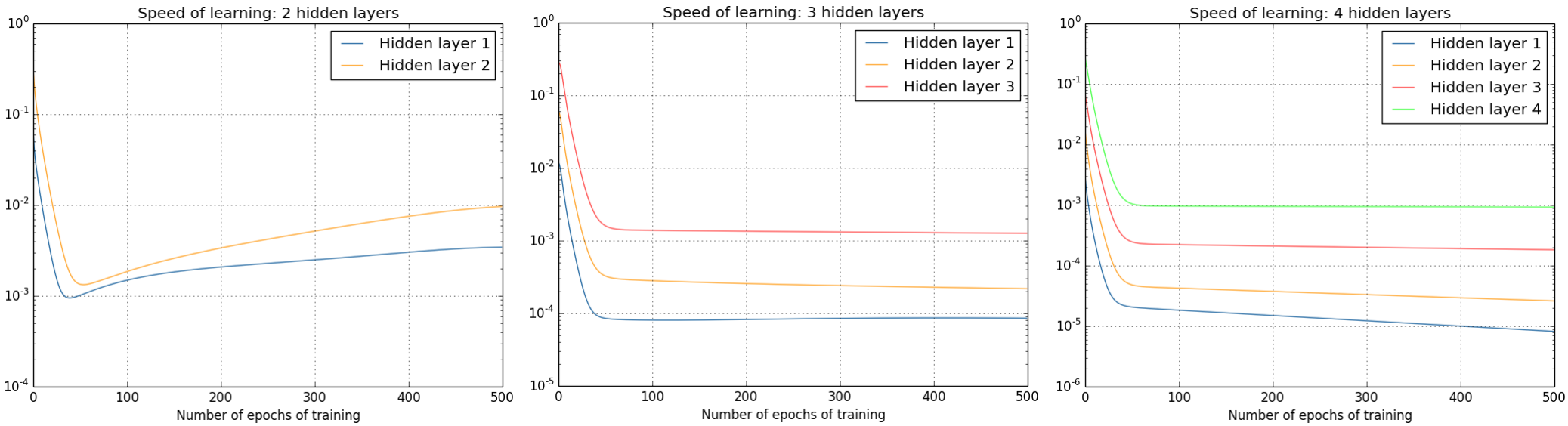
Przez długi okres stosowania jednokierunkowych ANN budowano głównie sieci **płytkie**, typowo z jedną warstwą ukrytą. Wynikało to z faktu, że sieć z jedną lub dwoma warstwami ukrytymi może zamodelować dowolną funkcję. Ponadto, uczenie sieci głębszych jest bardzo powolnym procesem.

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. Te wartości gradientu determinują jak szybko może uczyć się sieć:

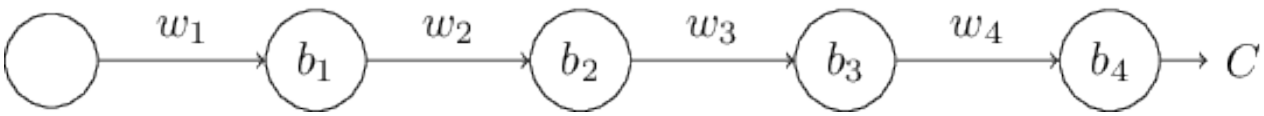


Okazuje się, że jest to zjawisko powszechne. W sieciach głębokich uczenie metodą wstecznej propagacji błędów z funkcją kosztu entropii krzyżowej gradienty kontrolujące szybkość zmian wag, determinujące szybkość uczenia, zanikają dramatycznie.

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:

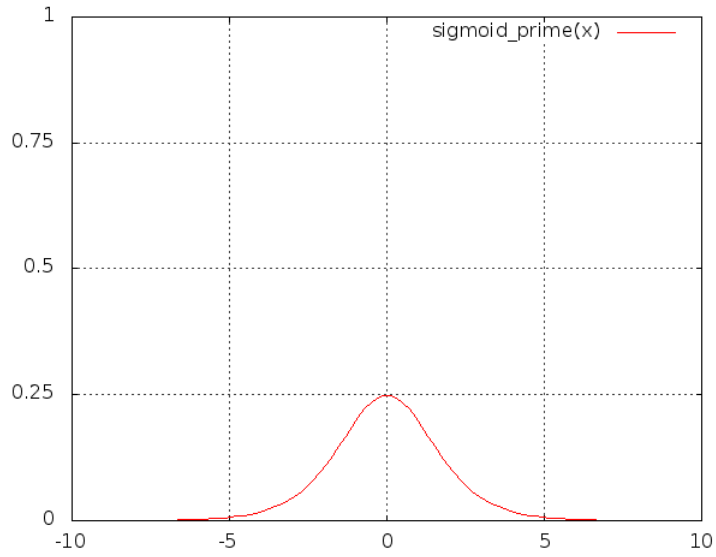
$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, w_2 \sigma'(z_2)\, w_3 \sigma'(z_3)\, w_4 \sigma'(z_4)\, \frac{\partial C}{\partial a_4}$$

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.

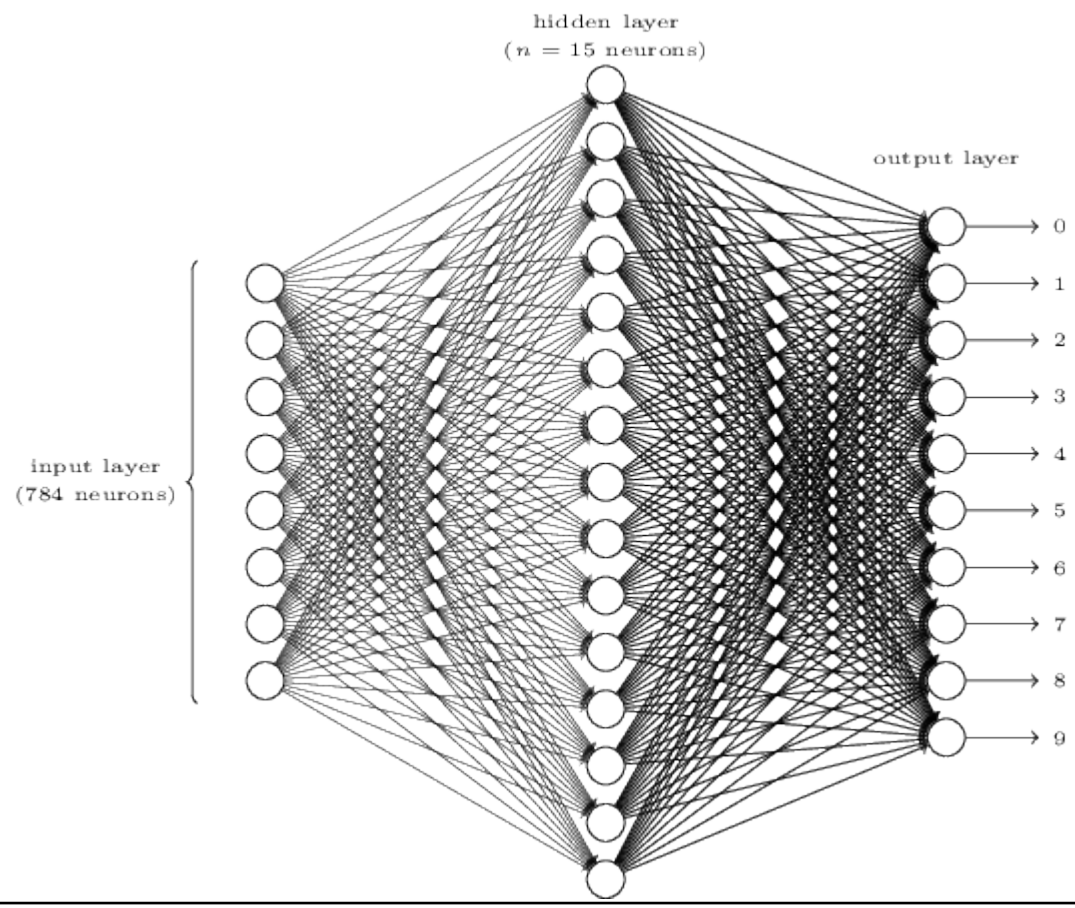
Istnieje szereg modeli sieci głębokich dających dobre efekty uczenia się. Jednym z nich są **sieci konwolucyjne**, typowo nadające się do przetwarzania obrazów. Wynika to z zasady działania operacji **konwolucji** przetwarzającej duży zbiór danych mniejszym filtrem, stopniowo przesuwającym się po zbiorze, 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, mapy cech, i agregacja obszarów (*pooling*).

Sieci konwolucyjne: dlaczego?

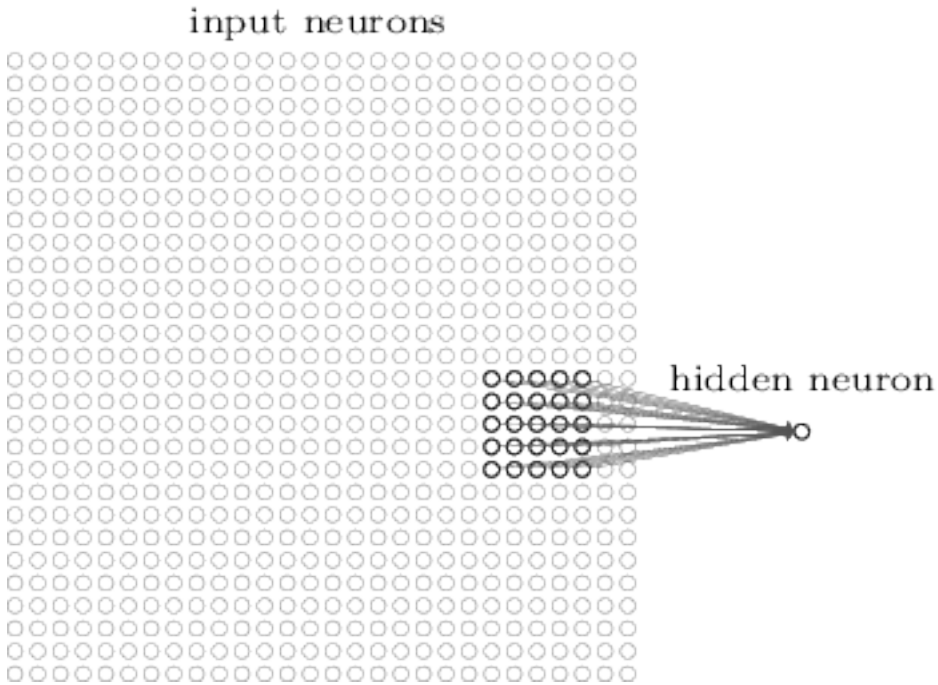
We wspomnianym wcześniej przykładzie analizy obrazów ręcznie pisanych cyfr (baza MNIST) o rozmiarach 28x28 pikseli zastosowana była sieć z pierwszą warstwą (wejściową) o rozmiarze 784 (=28x28) pikseli, z kompletem połączeń do wszystkich neuronów warstwy drugiej (ukrytej). Ma to sens z punktu widzenia 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*).

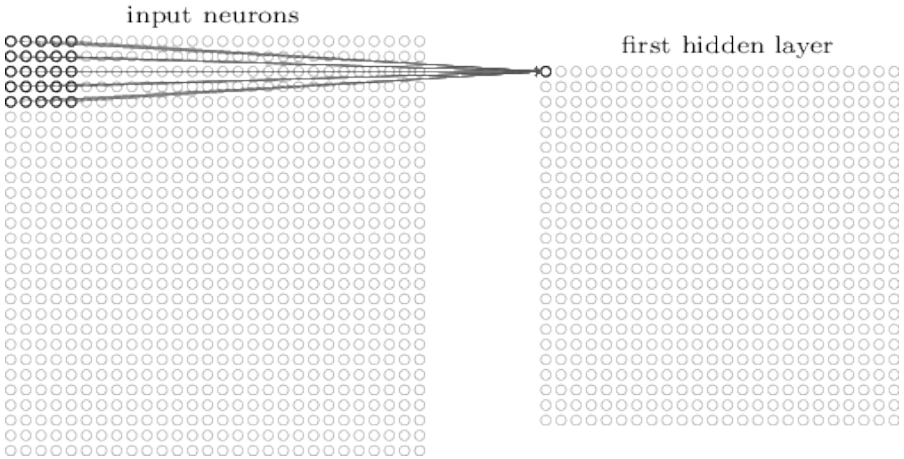


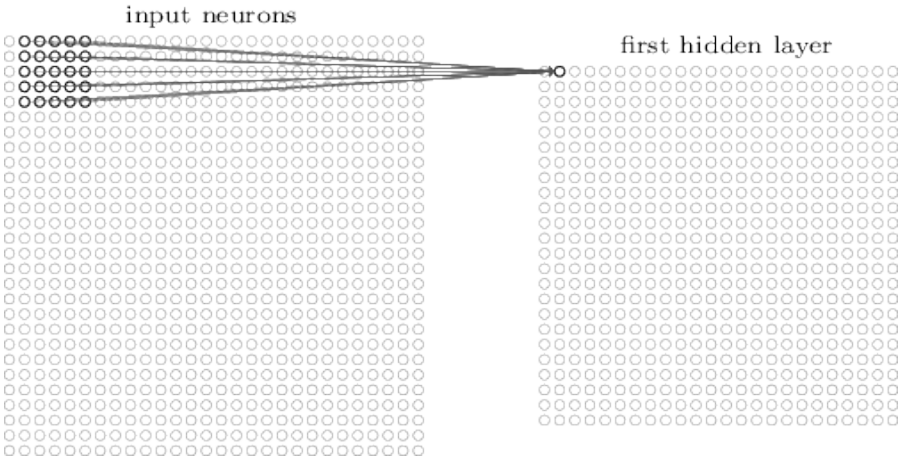
Pierwszą ukrytą warstwę sieci stanowić będzie warstwa konwolucji, przetwarzająca każde kolejne lokalne pole receptywne filtrem konwolucyjnym. Filtr jest niewielką tablicą (macierzą) przetwarzającą wszystkie neurony pola receptywnego, odwzorowującą je na pojedynczy neuron drugiej warstwy. Na przykład, macierz konwolucji może mieć rozmiar 5x5 jak na powyższym obrazku.

Przetwarzanie konwolucyjne

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

Przetwarzanie całego 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. Uwzględniając fakt, że przyłożenie filtra 5x5 wykonywane jest od brzegu z dwuelementowym marginesem, widać, że dla obrazu wejściowego o rozmiarze 28x28 pikseli obraz wynikowy powstający w warstwie konwolucji będzie miał rozmiar 24x24.

Arytmetyka konwolucji

W powyższym przykładzie macierz jądra konwolucji przesuwana była z krokiem 1. Zastosowanie większego kroku spowodowałoby powstanie w warstwie konwolucji mniejszego obrazu wynikowego. Np., dla kroku 2 byłby on około dwa razy mniejszy.

Ponadto, w tym przykładzie jądro konwolucji było przykładane w obrębie obrazu. To powoduje, że obraz wynikowy jest mniejszy o kilka pikseli, w zależności od rozmiaru tego jądra (dokładniej, jest mniejszy o rozmiar jądra minus jeden, np.: 28-(5-1)=24).

To rozwiązanie nazywane jest **przycinaniem obrazu**, ponieważ splot nie przetwarza bezpośrednio pikseli blisko krawędzi, i dla tych pikseli nie generuje wyników.
W przetwarzaniu obrazów dla potrzeb uczenia maszynowego jest to akceptowalne, ponieważ istotna jest treść wynikowego obrazu, a nie jego forma. Jednocześnie, to rozwiązanie nie wprowadza do obrazu żadnych zakłóceń ani obcych elementów.

Jest również możliwy inny schemat, gdzie jądro konwolucji przetwarza wszystkie elementy oryginalnego obrazka od brzegu do brzegu. To znaczy, przetwarzanie zaczyna się od przyłożenia elementu środkowego jądra splotu do piksela (1,1) obrazu źródłowego, czego wynikiem jest wygenerowanie obrazu wynikowego o rozmiarze identycznym do źródłowego. Jednak wtedy część jądra obejmuje obszar poza obrazem, i pytanie jak obliczać wynik splotu w takim przypadku.

Istnieje szereg możliwych rozwiązań przetwarzania brzegu obrazu jądrem splotu:

rozszerzanie — wartości elementów na krawędzi są powielane koncepcyjnie tak daleko, jak to konieczne, aby zapewnić wartości dla splotu; piksele narożne są powielane na cały obszar narożnika:

animacja: https://upload.wikimedia.org/wikipedia/commons/thumb/1/19/2D_Convolution_Animation.gif/375px-2D_Convolution_Animation.gif

zawijanie — obraz jest koncepcyjnie zawijany na przeciwległą krawędź

odbicie — obraz jest koncepcyjnie odbijany lustrzanie na krawędziach; np. próba odczytania piksela 3 jednostki poza krawędzią odczytuje piksel 3 jednostki wewnątrz obrazu

przycinanie jądra — w obliczaniu wartości splotu pomijane są składniki, które wykorzystują elementy spoza oryginalnego obrazu; to rozwiązanie wymaga odpowiedniego skalowania wyniku

wartości stałe — użycie stałej wartości dla pikseli spoza obrazu; na przykład może to być wartość średnia, taka jak szarość (127,127,127)

Współdzielenie wag warstwy konwolucyjnej

O ile w przetwarzaniu obrazów wykorzystywane są konkretne filtry konwolucyjne, o znanych własnościach, to w trenowaniu sieci konwolucyjnych filtr uczy się przez dostrajanie wag i *bias*u. Każdy obraz przetwarzany jest na całej powierzchni tym samym jądrem splotu, które stopniowo dostosowuje swoje wagi. Następnie to samo dzieje się z kolejnym obrazem serii uczącej, i dalej wiele razy ponownie na tych samych obrazach w ramach serii uczącej (*batch*).

W tym procesie cały czas wykorzystywane jest jednak to samo jądro splotu, które w przypadku rozmiaru 5x5 posiada 25 wag i jeden *bias*. W ten sposób zaledwie 26 parametrów obsługuje połączenie pomiędzy pierwszą i drugą warstwą sieci, i tylko te 26 parametrów intensywnie uczy się w całym procesie. Jest to zasadnicza różnica pomiędzy architekturą konwolucji a zwykłą architekturą sieci jednokierunkowej z kompletem połączeń pomiędzy tymi warstwami, które wymagałby prawie pół miliona wag (28x28x24x24) i 576 *bias*ów. Dzięki temu uczenie sieci może być efektywne.

Ten efekt określany jest jako **współdzielenie wag** warstwy konwolucji.

Mapy cech

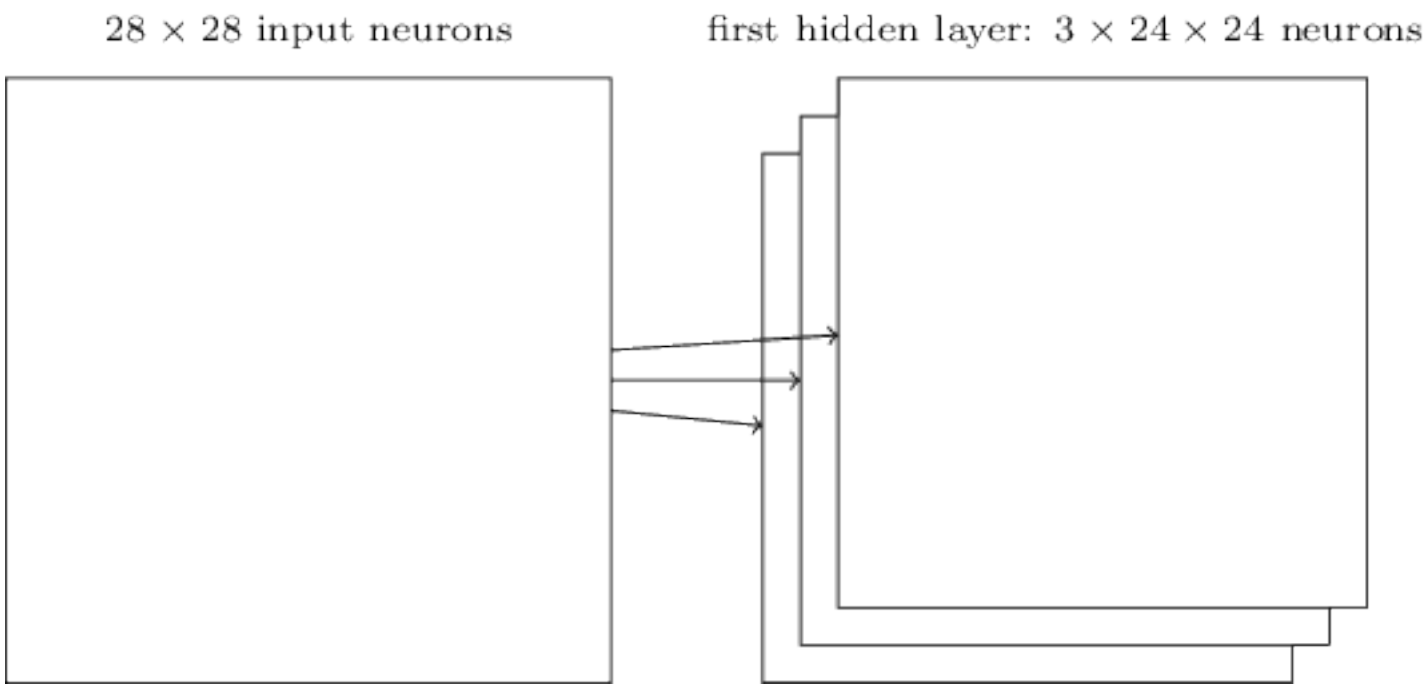
W efekcie procesu trenowania warstwy konwolucji, którą właśnie utworzyliśmy, nauczy się ona rozpoznawać pewną cechę w obrazie. Ta cecha może mieć prostą interpretację, taką jak centralna plama lub krawędź, może być cechą liniową biegnącą pod pewnym kątem, albo cechą obszarową lub punktową, ale może też być jakimś trudnym do opisania wzorcem (choć musi on być prosty, ze względu na rozmiar filtra).

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

Prostokątna struktura (obraz) wygenerowana przez filtr konwolucji nazywana jest **mapą cech**.

Wiele filtrów konwolucyjnych

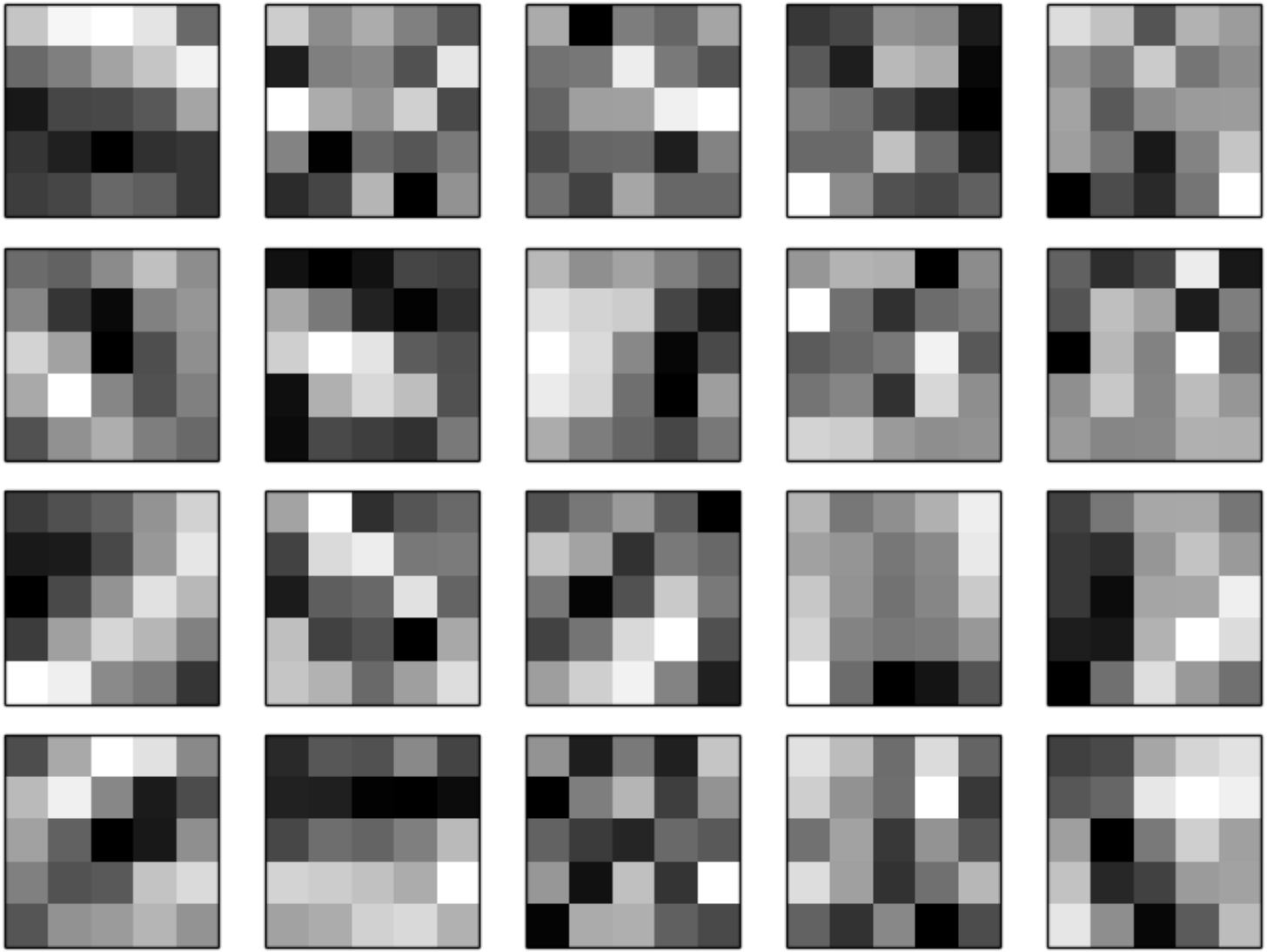
Warstwa splotu uczy się rozpoznawać losową cechę obrazu. Ta cecha może być mniej lub bardziej przydatna w interpretacji całego obrazu. Możemy zmaksymalizować szansę na uzyskanie przydatnych filtrów wykrywających cechy, tworząc ich kilka lub nawet wiele. Dzięki randomizacji, można oczekiwać, że każdy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

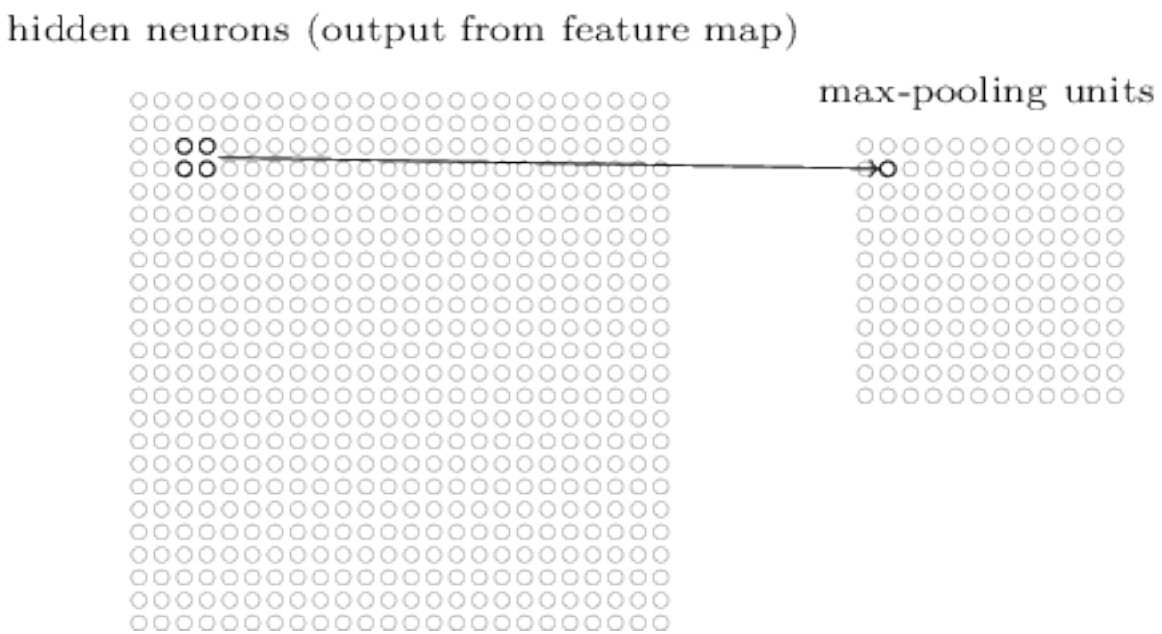
Wiele filtrów konwolucyjnych (cd.)

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



Warstwy agregacji

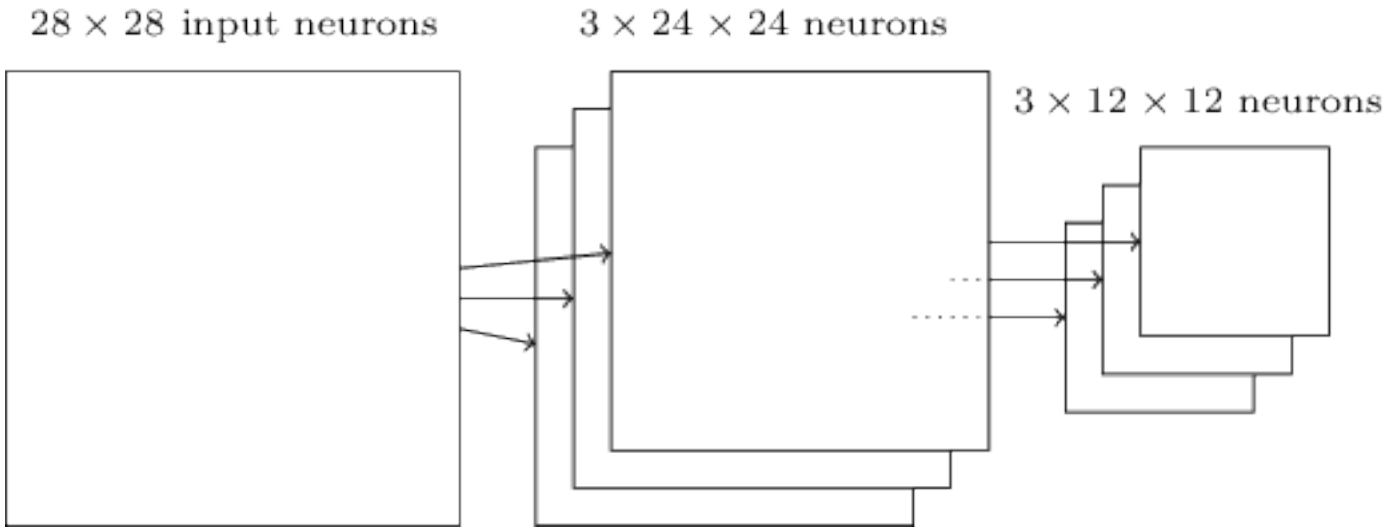
Mapy cech utworzone w warstwie splotu reprezentują rozpoznane drobne cechy oryginalnego obrazu. Każda z nich jest jednak prawie tak duża, jak obraz oryginalny (zależy to od wielkości kroku, plus marginesów wynikających z wielkości filtra splotu). Można powiedzieć, że mapy cech są tworzone w pełnej rozdzielczości 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.)

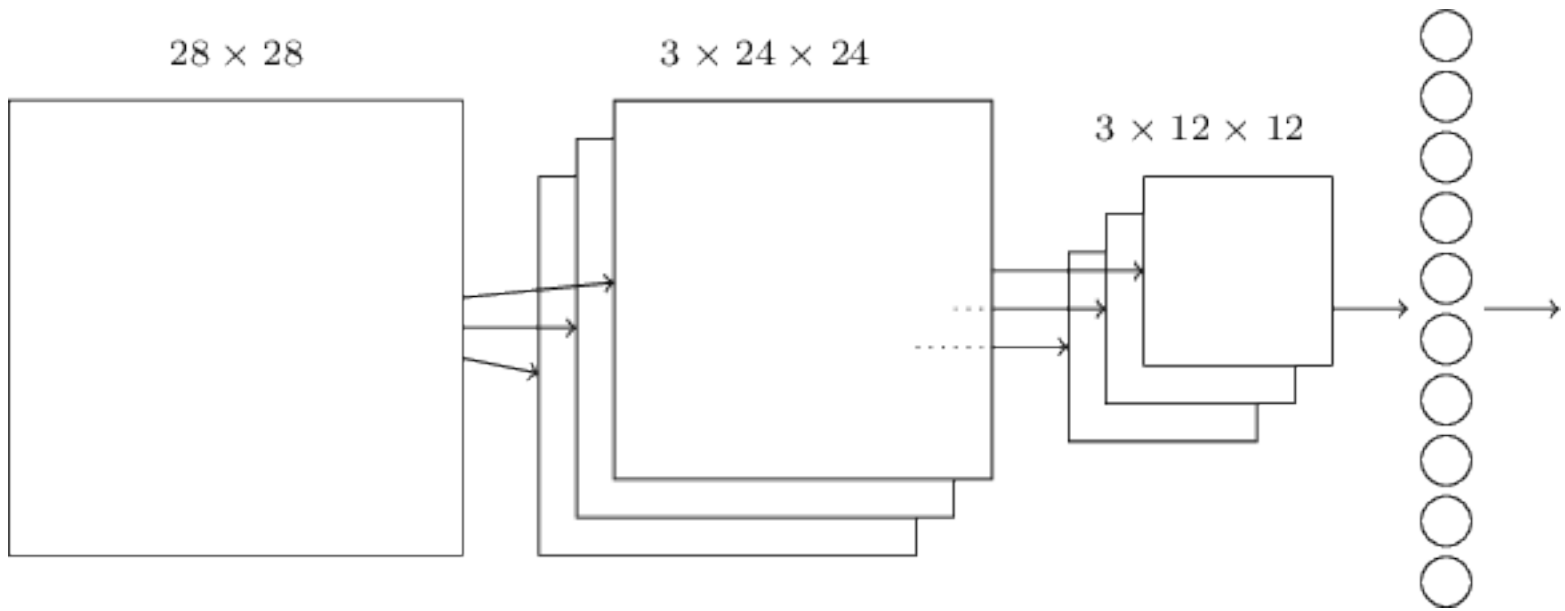
W prezentowanym przykładzie zastosowano agregacji 2x2 obszarów neuronowych. Odpowiada to zmniejszeniu rozmiaru piksela obrazu o połowę, z 24x24 do 12x12. Każda mapa obiektów z warstwy konwolucji powinna być agregowana w ten sam sposób, co daje następującą strukturę:



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

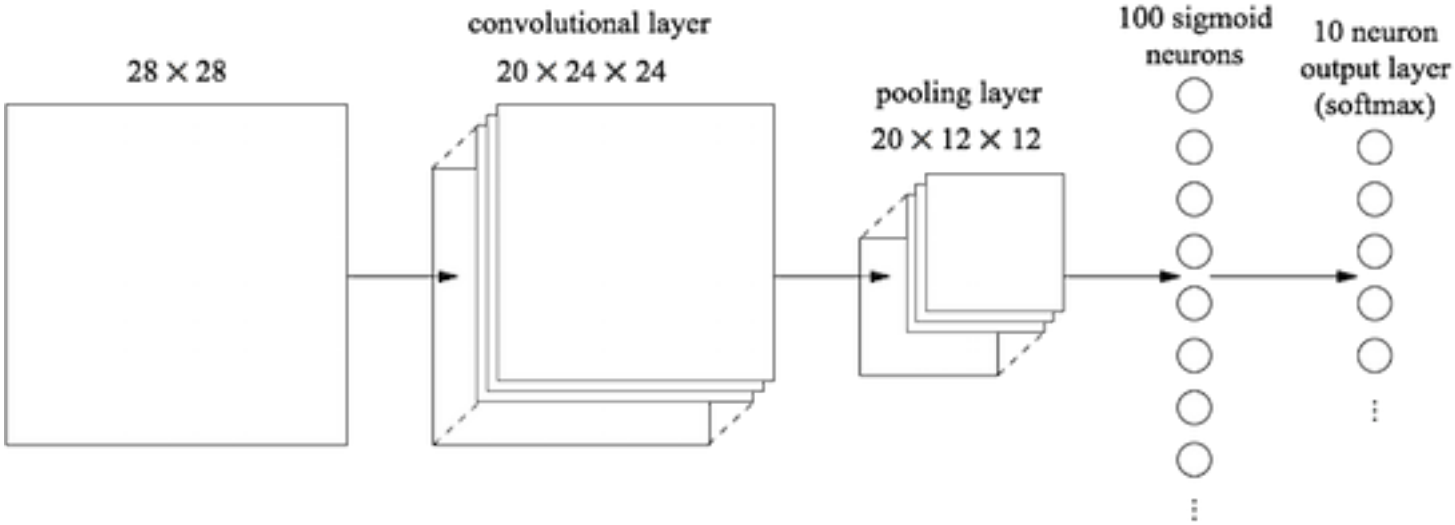
Jeśli uznamy, że utworzone do tej pory warstwy sieciowe powinny wystarczyć do rozpoznania odręcznych cyfr, możemy zakończyć dodając warstwę wyjściową składającą się z dziesięciu neuronów, z których każdy wytrenowany jest do odpowiedzi na inną cyfrę, tak jak poprzednio:



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.

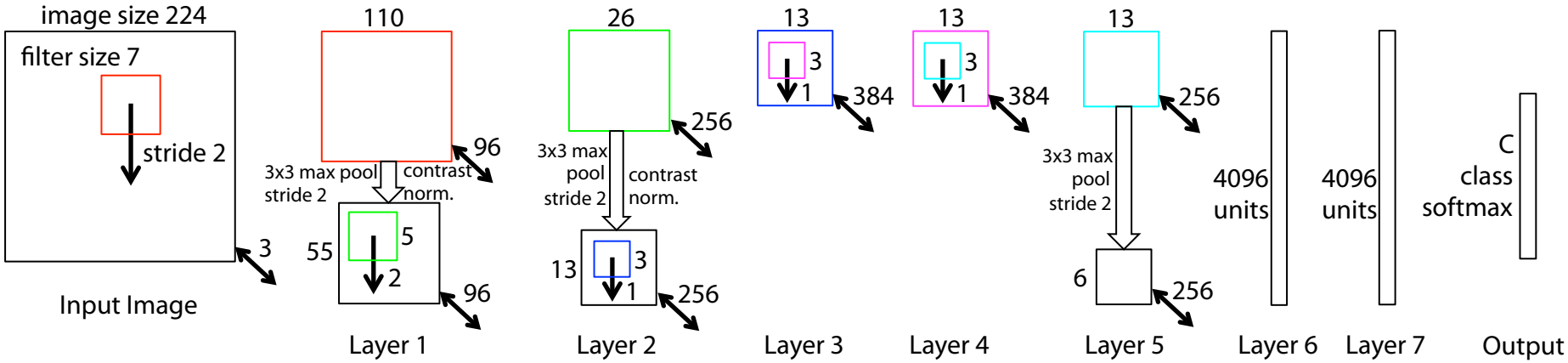
Inne warianty

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.

Rozbudowana sieć konwolucyjna

Przykład 8-warstwowej sieci konwolucyjnej do rozpoznawania obrazów zbioru ImageNet 2012 (1.3 miliona obrazów z ponad 1000 klasami obiektów)[1]:



W warstwie wejściowej sieci obraz o rozmiarach 224x224 z 3 warstwami RGB jest przetwarzany 96 różnymi filtrami splotu 7x7 z krokiem 2. Następnie powstałe 96 map cech o rozmiarach 110x110 (czerwone) są w warstwie 1: (i) przepuszczane przez filtr ReLU (nie pokazano), (ii) agregowane funkcją max o rozmiarze 3x3 z krokiem 2, i (iii) normalizowane kontrastowo, tworząc 96 nowych map cech o rozmiarach 55x55. Następnie podobne operacje są powtarzane w warstwach 2, 3, 4, 5. Dwie ostatnie warstwy (6 i 7) są w pełni połączone, przyjmując na wejściu 256 map 6x6 (łączny wymiar 6x6x256=9216). Warstwa wyjściowa realizuje funkcję softmax z C klasami.

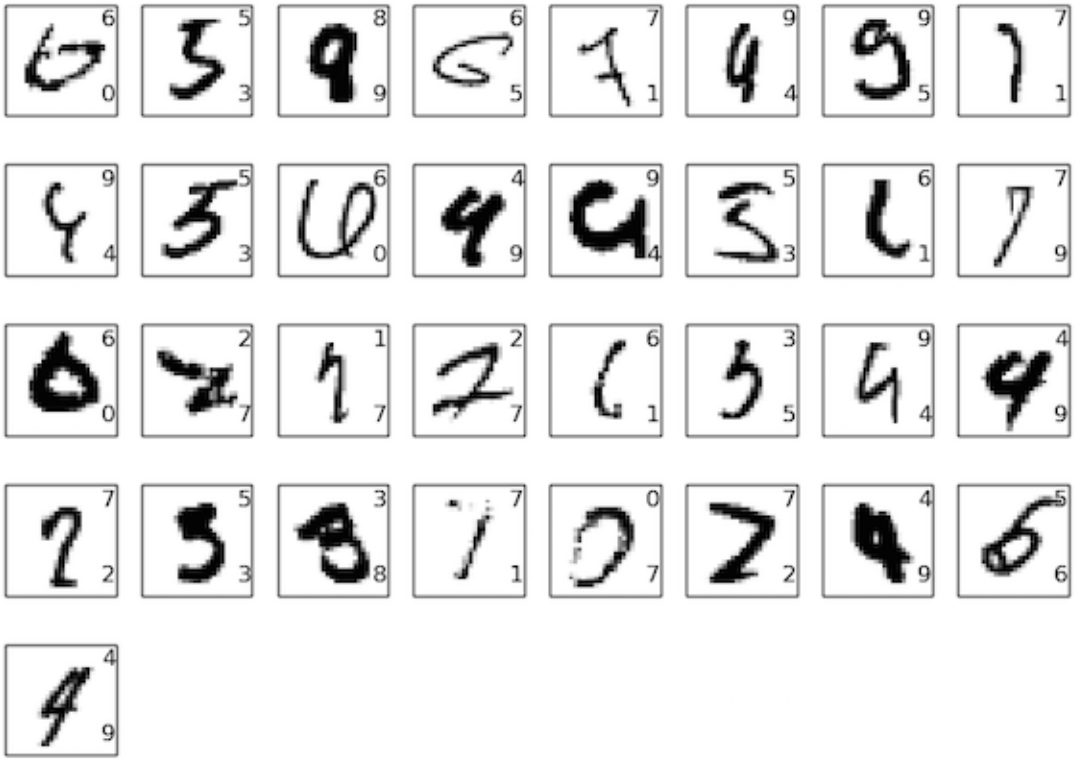
[1] Matthew D. Zeiler and Rob Fergus: Visualizing and Understanding Convolutional Networks, w: Computer Vision and Pattern Recognition, Computer Vision - ECCV 2014, Lecture Notes in Computer Science, vol 8689, Springer

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.

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



Zastosowania sieci konwolucyjnych

Przetwarzanie obrazów jest ważnym i szerokim obszarem zastosowań sieci konwolucyjnych, ale nie jedynym. Obszary zastosowań:

1. Przetwarzanie obrazów i wideo:

   klasyfikacja obrazów, wykrywanie obiektów na obrazach, śledzenie obiektów na wideo, wykrywanie ruchu, nadzorowanie ruchu

2. Przetwarzanie języka naturalnego:

   klasyfikacja tekstów, tłumaczenie tekstów

3. Zastosowania w ochronie zdrowia:

   analiza obrazów medycznych, diagnozowanie i przewidywanie chorób

4. Inne zastosowania:

   automatyczne prowadzenie pojazdów, usługi finansowe, handel i *e-commerce*, automatyka przemysłowa, itp.

Sieci rekurencyjne

Rekurencyjne sieci neuronowe (*Recurrent Neural Networks RNN*) w odróżnieniu od sieci jednokierunkowych zawierają cykle, to znaczy sygnał generowany przez sieć jako wyjście w pewnej warstwie jest przekazywany jako wejście do warstwy wcześniejszej. Zakładamy również, że występuje opóźnienie czasowe przekazania tego sygnału. To znaczy, jednostka może otrzymać na wejściu wartość obliczoną z jej sygnału wyjściowego wygenerowanego we wcześniejszym kroku obliczeniowym.

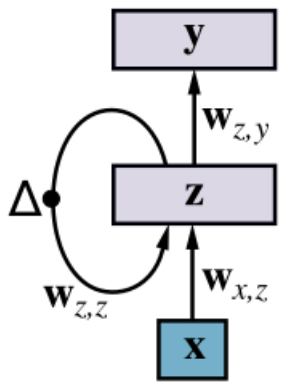
Oznacza to, że sieci RNN mogą posiadać **pamięć**, ponieważ sygnały otrzymane wcześniej mogą wpływać na działanie sieci w kroku bieżącym. To z kolei oznacza, że takie sieci mogą wykonywać obliczenia o dość ogólnym charakterze, ponieważ czyni je to podobnymi do prostych programów komputerowych.

Tutaj rozważymy jedynie zastosowania sieci RNN do analizy danych sekwencyjnych, czyli takich gdzie w każdym kroku czasowym pojawia się nowy wektor wejściowy $\mathbf{x}_t$. W budowie takich sieci stosowane jest **założenie Markowa**, które mówi, że ukryty stan sieci $\mathbf{z}_t$ w pełni determinuje dalsze zachowanie sieci, niezależnie od sekwencji wcześniejszych sygnałów wejściowych. Zakładamy, że ten wewnętrzny stan podlega aktualizacji przez pewną funkcję $f_{\mathbf{w}}$ parametryzowaną przez wektor wag $\mathbf{w}$ ustalanych w procesie uczenia sieci:

$$\mathbf{z}_t = f_{\mathbf{w}}(\mathbf{z}_{t-1}, \mathbf{x}_t)$$

Trenowanie sieci RNN

Rozpatrujemy model sieci RNN widoczny na rysunku poniżej. Zawiera on warstwę wejściową $\mathbf{x}$, warstwę ukrytą $\mathbf{z}$ z połączeniami rekurencyjnymi, i warstwę wyjściową $\mathbf{y}$:



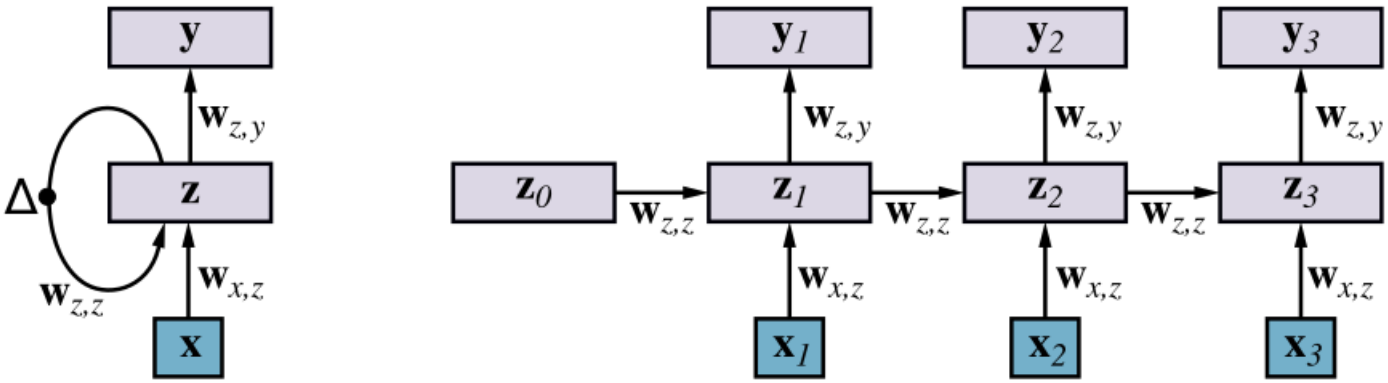
Ten model opisany jest równaniami:

$$\begin{aligned} \mathbf{z}_t &= f_{\mathbf{w}}(\mathbf{z}_{t-1}, \mathbf{x}_t) = \mathbf{g}_z(\mathbf{w}_{z,z}\mathbf{z}_{t-1} + \mathbf{w}_{x,z}\mathbf{x}_t) \\ \hat{\mathbf{y}}_\mathbf{t} &= \mathbf{g}_y(\mathbf{w}_{z,y}\mathbf{z}_t) \end{aligned}$$

W tych wzorach $\mathbf{g}_z$ i $\mathbf{g}_y$ są funkcjami aktywacji. Wzory te nie zawierają *bias*ów, ponieważ zakładamy, że zastąpione one zostały przez dodatkowe sztuczne wejście o wartości $+1$ dla każdej jednostki, z wagą która zastępuje *bias*.

Trenowanie sieci RNN (cd.)

Mając dane warstw $\mathbf{x}$ i $\mathbf{y}$ z kolejnych kroków czasowych t stanowiące dane uczące sieci, możemy „rozwinąć" ten model z rysunku po lewej dla T kroków czasowych, jak widać na rysunku po prawej. Zauważmy, że wagi $\mathbf{w}_{x,z}$, $\mathbf{w}_{z,z}$, i $\mathbf{w}_{z,y}$, są współdzielone dla kolejnych kroków czasowych.



Posługując się modelem sieci RNN rozwiniętym przez kroki czasowe można wyprowadzić wzory na pochodne funkcji straty po wagach dla poszczególnych warstw. Pozwala to skonstruować algorytm podobny do algorytmu propagacji wstecznej dla sieci jednokierunkowej i wykorzystać go do uczenia sieci rekurencyjnej. Ten algorytm nazywa się **propagacją wsteczną przez czas** (*backpropagation through time*).

Długoterminowa pamięć krótkotrwała LSTM

Algorytm propagacji wstecznej przez czas jest podatny na zjawiska zanikającego i eksplodującego gradientu, podobnie jak podstawowy algorytm propagacji wstecznej. Jednocześnie zdolność reakcji na stan wygenerowany wyłącznie w jednym wcześniejszym kroku stanowi poważne ograniczenie limitujące zastosowania sieci RNN w wielu dziedzinach.

Dla rozwiązania tych problemów powstała wyspecjalizowana architektura RNN, zwana **długoterminową pamięcią krótkotrwałą**[2] (*long short-term memory LSTM*).

Sieć LSTM pozwala na zapamiętywanie napotkanych sygnałów na wiele cykli, dzięki specjalnemu mechanizmowi kontroli pamiętanych wartości. Wykorzystuje ona wektor stanu, podobnie jak zwykła sieć RNN, stanowiący w tym wypadku pamięć krótkotrwałą, podlegającą normalnej aktualizacji w kolejnych krokach czasowych. Jednak w dodatku do niej LSTM posiada pamięć długoterminową, która jest przekazywana w kolejnych krokach czasowych, i może być zapomniana tylko w określonych warunkach, pod kontrolą specjalnej **bramki zapominania**.

[2] W polskiej wersji podręcznika Russella i Norviga określenie to zostało przetłumaczone jako „długotrwała pamięć krótkoterminowa", co jednak nie zgadza się z ogólnie istniejącą polską terminologią.

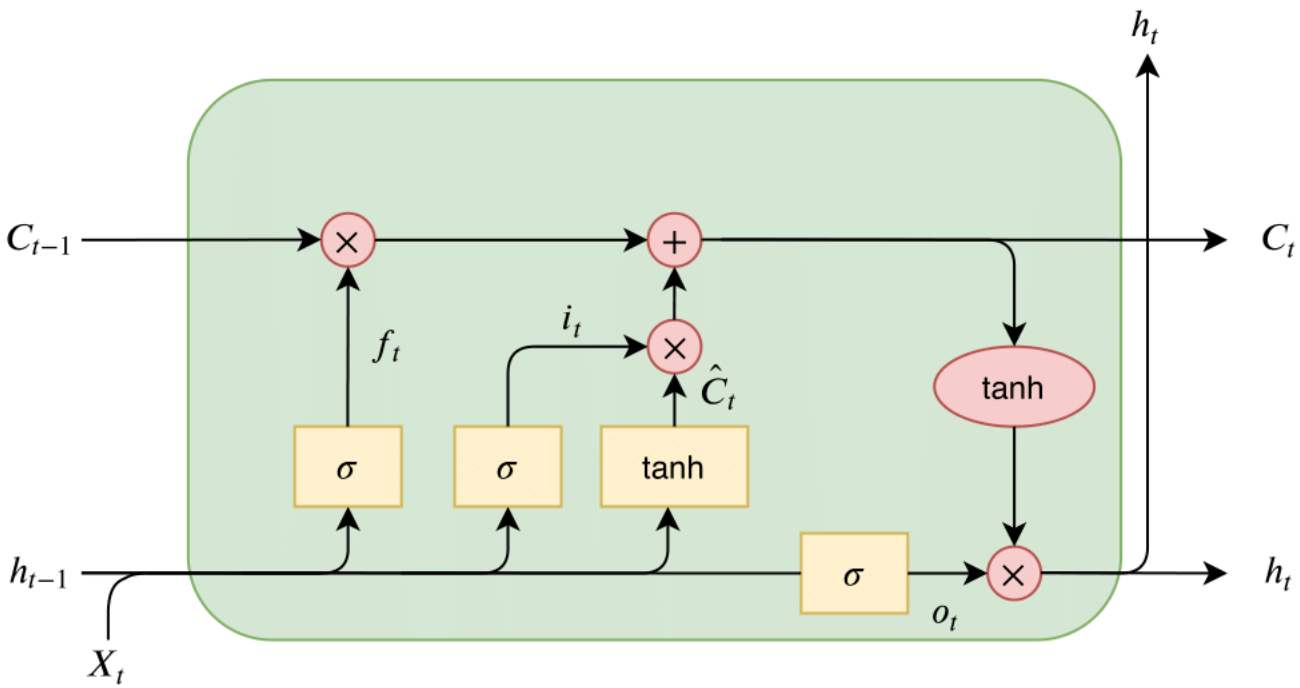
Jednostka LSTM

Jednostka sieci LSTM zamiast pojedynczej funkcji aktualizacji posiada trzy **bramki**:

bramka zapominania (*forget gate*) decyduje czy informacja długoterminowa c_t z poprzednich kroków ma być zachowana czy zapomniana,

bramka wejściowa (*input gate*) decyduje jakie nowe informacje należy przechować w komórce; wykorzystuje dwie funkcje przejścia: sigmoidalną, która decyduje które wartości należy aktualizować, i funkcję tanh tworzącą wektor nowych wartości, które będą dodane do wartości stanu długoterminowego c_t komórki,

bramka wyjściowa (*output gate*) generuje nowy ukryty stan komórki, który określa wartość wyjścia h_t na podstawie pamięci c_t zaktualizowanego stanu ukrytego.



Literatura i materiały pomocnicze

1. Stuart Russell, Peter Norvig: Sztuczna inteligencja. Nowe spojrzenie, Wydanie IV, Tom 2, Wyd.Helion, 2023, rozdział 21.

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

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