Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Przemysław Chlipała

Nr albumu: 1148609

Metody wizualizacji optymalizacji aktywacji neuronów sieci konwolucyjnych

Praca magisterska
na kierunku Infromatyka Stosowana

Praca wykonana pod kierunkiem
dr hab. Tomasz Kawalec
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Kraków 2019

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

.....	.....
Kraków, dnia	Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

.....	.....
Kraków, dnia	Podpis kierującego pracą

Spis treści

1. Wstęp

Rozwój nauki oraz stały wzrost mocy obliczeniowej komputerów pozwolił na rzeczy, które wydawałby się niemożliwe jeszcze dekadę temu. Chociaż pierwsze ważne dla sieci neuronowych koncepty sięgają lat pięćdziesiątych, a kluczowe kwestie takie jak algorytm propagacji wstecznej lat siedemdziesiątych, to dopiero w ciągu ostatnich dziesięciu lat nastąpiła popularyzacja tej technologii na nie spotykaną dotąd skalę.

Przy pomocy sieci neuronowych zaczęto budować systemy przewidujące ceny nieruchomości, indeksy giełdowe czy to jakie jest prawdopodobieństwo na to, że ktoś zapadnie na przewlekłą chorobę. Co więcej, wiele firm zbudowało na tych modelach ogromny kapitał. Przykładowo, sieci posłużyły im do predykcji zainteresowania danym produktem. W 2014 roku Internet obiegła informacja, że firma Amazon przewiduje to co ich klient zamierza kupić zanim jeszcze złoży zamówienie. Między innymi, dzięki temu udało im się skrócić czas oczekiwania na zamówienie do 48 godzin.

Ale na przewidywaniach się nie skończyło. Rekurencyjne sieci neuronowe rozpoznają słowa kluczowe i sprawiają, że możemy wydawać polecenia naszym urządzeniom. Co więcej, z ich pomocą tłumaczenia prosto z internetowego translatora brzmią z dnia na dzień coraz bardziej naturalnie. Automatyczne transkrypcje są coraz szerzej dostępne pod filmami udostępnionymi w Internecie a ich poprawność rośnie z dnia na dzień.

Komputery zaczęły rozpoznawać przedmioty zapisane na wideo i zdjęciach. Ta futurystyczna technologia wspomaga rzeczy tak prozaiczne jak kontrola jakości czipsów na liniach produkcyjnych. Automatyczne sortownie odpadów mieszanych przynoszą nieoceniony zysk dla środowiska i realny, liczony w dolarach zysk dla ich właścicieli.

W mojej pracy skupię się na sieciach z dziedziny przetwarzania obrazu. Przy pomocy neuronowych sieci konwolucyjnych (z angielskiego Convolutional Neural Networks – CNN) wpierw sklasyfikuję kilka prostych symboli przy pomocy sieci konwolucyjnej LeNet-5, a następnie zwizualizuję, czego tak naprawdę nauczył się mój automat.

Przybliżę również architekturę sieci VGG-16 a wraz z nią technikę NSS (z angielskiego Neural Style Transfer). Przy jej pomocy zwizualizuję cechy nauczone przez model.

2. Gęste sieci neuronowe

2.1. Algorytmy uczące się

W przypadku każdego wariantu sieci neuronowej mówimy o modelu luźno inspirowanym biologicznym odpowiednikiem. Jego zadaniem jest rozwiązanie pewnej zadanej klasy problemu T. Proces ten nazywamy nauką lub treningiem. Nauka została zdefiniowana przez prof. Toma Mitchella w następujący sposób: "Program komputerowy uczy się na podstawie doświadczenia E, w związku z pewną klasą problemów T przy uwzględnieniu miary skuteczności P, jeżeli jego sprawność w wykonywaniu zadań T, wyrażona przy pomocy P rośnie razem z doświadczeniem E."

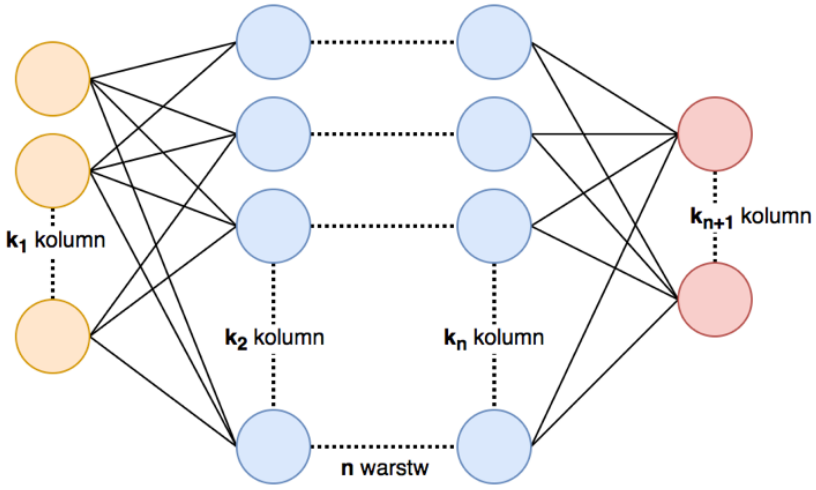
Proces treningu w przypadku modeli przytoczonych w tej pracy będzie polegał na ustaleniu wag przy poszczególnych węzłach sieci neuronowej, dalej nazwanych neuronami. Składać się on będzie z dwóch części: propagacji wprzód oraz propagacji wstecznej.

Praca traktuje o zagadnieniach z dziedziny uczenia z nauczycielem (z angielskiego Supervised Learning). W związku z tym, podczas treningu, propagacja wprzód będzie polegała na obliczeniu wartości poprzez sieć dla danych uprzednio sklasyfikowanych i oznakowanych przy pomocy innego automatu lub człowieka. Uzyskany wynik jest początkowo losowy i zostaje użyty do obliczenia wartości funkcji kosztu. Na podstawie funkcji kosztu, wagi sieci zostaną skorygowane w procesie propagacji wstecznej.

2.2. Gęsta sieć neuronowa

Podstawową implementacją sieci neuronowej jest tzw. sieć w pełni połączona (oznaczam FC z angielskiego Fully Connected Network). Składa się z warstw: wejściowej, jednej lub więcej warstw ukrytych i warstwy wyjściowej.

W przypadku gdy model zawiera stosunkowo dużą ilość warstw ukrytych, na schemacie oznaczonych jako n, mówimy o głębokich sieciach neuronowych. Ten wariant sieci neuronowej charakteryzuje się tym, że każdy węzeł jej warstwy jest połączony z każdym. Warstwa

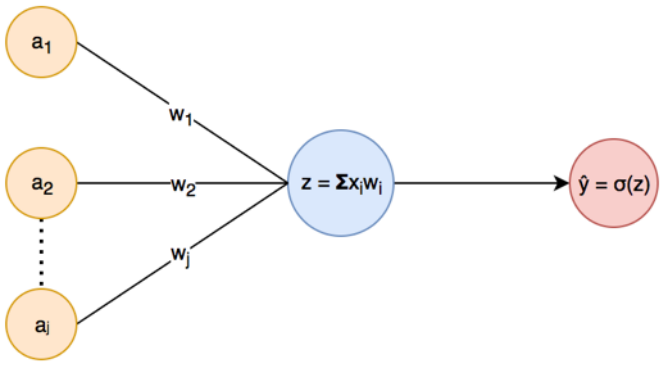


Rysunek 2.1: Schemat w pełni połączonej sieci neuronowej.

wejściowa, oznaczona kolorem żółtym, doprowadza dane wejściowe do warstw ukrytych oznaczonych kolorem niebieskim. Warstwa wyjściowa oznaczona jest kolorem czerwonym.

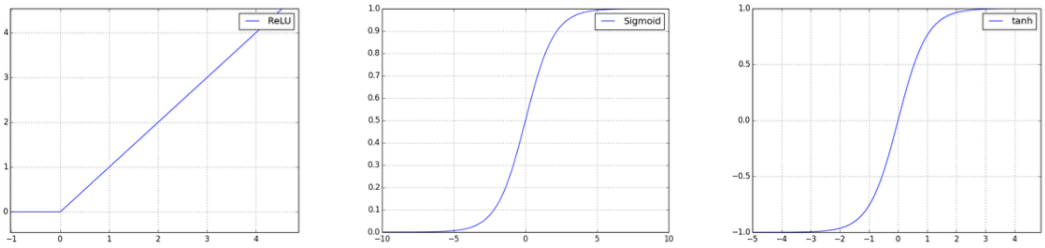
2.3. Neuron gęstej sieci neuronowej

Pojedynczy neuron można przedstawić następującym schematem.



Rysunek 2.2: Schemat w pełni połączonej sieci neuronowej.

Wartości a_1 do a_j reprezentują wartości z warstwy sieci neuronowej a w szczególności wartości początkowe $x_1 ... x_j$. Każda z nich jest przemnożona przez odpowiadającą jej wagę w i zsumowana. Następnie, tak uzyskaną wartość używamy do wyliczenia ρ nazwanego funkcją aktywacji neuronu. Zadaniem ρ jest wprowadzenie elementu nieliniowości. Mniej formalnie, ma na celu ustalenie czy neuron powinien zostać aktywowany czy też nie. Do najpopularniejszych funkcji aktywacji należą sigmoid, ReLU i tangens hiperboliczny.



Rysunek 2.3: Wybrane funkcje aktywacji, od lewej ReLU, sigmoid i tangens hiperboliczny.

2.4. Propagacja wprzód

By ustalić wartość neuronu posługujemy się następującym wzorami.

$$z_n^{[l]} = \sum_{i=1}^{j} w_{\text{ni}}^{[l]} a_i^{[l-1]} + b_n^{[l]}$$

$$a_n^{[l]} = \sigma(z_n^{[l]})$$

Gdzie:

- ρ to funkcja aktywacji,
- $_{(n)}$ oznacza numer neuronu w warstwie,
- $^{[l]}$ to numer warstwy,
- b to tak zwany bias. Jego wartość jest ustalana razem z wagami w procesie propagacji wstecznej,
- $w_{\text{ni}}^{[l]}$ jest to **i**-ta waga neuronu numer **n**, warstwy **l**,
- $a_i^{[l-1]}$ jest to wartość **i**-tego neuronu warstwy [l-1], w szczególnym przypadku może być to jedna z wartości wejściowych x.

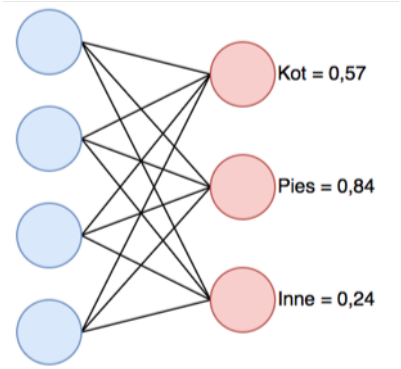
Używając tych wzorów, możemy obliczać wartości wag kolejnych warstw sieci aż do warstwy ostatniej. Tam w zależności od postawionego problemu, możemy wyliczyć kolejne aktywacje przy pomocy ρ (tylko w odróżnieniu od warstw ukrytych nie stosuje się tam ReLU, tylko np. sigmoid) lub możemy użyć funkcji softmax:

$$S(y_i) = \frac{e^{y_{\text{i}}}}{\sum_j^{y_n} e^{y_j}}$$

Jest to funkcja, która przyjmuje y_n elementowy wektor składający się z liczb rzeczywistych i przeprowadza go w rozkład prawdopodobieństwa składającego się z y_n elementów. Przydatną jej cechą jest to, że zachowuje proporcjonalny udział wartości w wynikowym prawdopodobieństwie – tj. elementy o wyższej wartości mają przypisane wyższe wartości prawdopodobieństwa.

Na przykładzie klasyfikacji, by uzyskać wynik, który można zinterpretować należy posłużyć się następującym sposobem postępowania. Niech przedostatnia warstwa $a^{[L-1]}$ sieci posiada n węzłów, których ilość jest równa ilości klas rzeczy, które chcemy rozpoznawać. Przyjmijmy również, że dana rzecz nie może należeć do dwóch klas równocześnie.

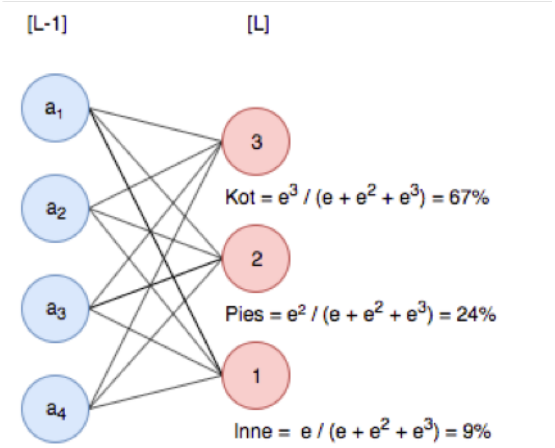
W przypadku aktywacji przy pomocy sigmoid uzyskujemy algorytm nazywany "jeden kontra każdy"(z ang. one vs all). Każdy z węzłów $a_i^{[L-1]}$ jest wejściem dla σ. Po wyliczeniu wartości, przyjmujemy próg z przedziału [0, 1], by uzyskać wynik dla każdego neuronu z osobna.



Rysunek 2.4: Przykładowy wynik 1 vs all dla trzech klas ze zbioru $\{$Kot, Pies, Inne.$\}$

By zobrazować działanie algorytmu posłużę się przykładem. Załóżmy, że zbudowano klasyfikator, mający na celu przypisanie danej postaci klasy ze zbioru kot, pies, inne. Każdą aktywacje z warstwy L-1 przeprowadzamy z osobna, przy pomocy sigmoid w wartość z przedziału od 0 do 1. Następnie, przyjmujemy próg, powyżej którego mówimy, że sieć rozpoznała daną klasę. Przykładowo, w przypadku z rysunku 4 przyjmując próg równy 0.5 mamy dwa możliwe wyniki. By wyeliminować jeden z nich możemy zwiększyć próg do 0.6 lub wybrać neuron z wyższą wartością. W obu przypadkach będzie to neuron oznaczający to, że klasyfikator rozpoznał psa.

Istnieją inne metody aktywacji warstw wyjściowych, każda z nich, która zostanie użyta w tej pracy będzie trzymała się podobnego schematu. Wynikiem sieci gęsto połączonej będzie wektor zer i jedynek, w którym 1 ustawione na odpowiednim miejscu w rzędzie będzie oznaczało, że dany przedmiot został rozpoznany. W przypadku rozrysowanym na rysunku 5 będzie to wektor [1 0 0].



Rysunek 2.5: Przykładowy wynik z użyciem softmax.

2.5. Funkcja kosztu

W przypadku problemów poruszanych w tej pracy, funkcję kosztu należy rozumieć jako przyporządkowanie wynikom uzyskanym przy pomocy sieci neuronowej wraz ze znanymi poprawnymi odpowiedziami, liczby reprezentującej „koszt". Wartość kosztu należy interpretować jako skuteczność w odgadywaniu poprawnych wyników w taki sposób, że im niższy koszt tym wyższa poprawność predykcji. Można powiedzieć, że sieć neuronowa próbuje znaleźć najwierniejsze przybliżenie oryginalnej, nieznanej funkcji nazywanej hipotezą, a celem funkcji kosztu jest modelowanie różnicy między przybliżeniem a funkcją, której próbujemy nauczyć sieć.

Istnieje wiele różnych funkcji kosztu. Opiszę tutaj tylko funkcję „cross entropy loss", która jest często wybierana przy treningu sieci, które na ostatniej warstwie używają softmax.

$$\frac{1}{m}\sum_{i=1}^{m}[y_i \log \acute{y}_i + (1-y_i)\log(1-\acute{y}_i)]$$

Gdzie:

– m – ilość klasyfikowanych przykładów,

– y_i – wartość oczekiwana dla przykładu numer i

– $\acute{y}_i$ – wartość uzyskana przy pomocy sieci dla przykładu numer i,

Jest to suma błędów pojedynczych predykcji dla wszystkich przykładów dostępnych w danym zestawie danych treningowych. Zadaniem treningu, będzie znalezienie jej minimum dla danego problemu. By udało się to sprawnie zrobić, będę korzystał z faktu, że ta i inne funkcje kosztu którymi będę się posługiwał są różniczkowalne i ciągłe.

2.6. Propagacja wstecz

Jest to modyfikacji wag sieci neuronowej w z uwzględnieniem funkcji kosztu $J(w)$. Popularnym sposobem jest metoda gradientu prostego (częściej spotykana w wariacji z tzw. pędem, z angielskiego "Gradient descent with momentumłub dalsze jej modyfikacje jak np. algorytm adam). W swej najprostszej wersji można sprowadzić go do następujących kroków:

1. Wybierz punkt startowy w_0.
2. $w_{k+1} = w_k - \alpha \nabla f(w_k)$.
3. Jeżeli $\|w_{k+1} - w_k\| \geq \varepsilon$ idź do 2.
4. Koniec.

Gdzie:

- α – współczynnik uczenia, zazwyczaj niewielka ($\alpha < 0.01$) dodatnia liczba rzeczywista. Podczas treningu sieci neuronowych często $\alpha = const$, choć sam algorytm gradientu prostego posiada wariację ze zmienną wartością α.
- $\nabla f(w_k)$ – jest to gradient funkcji reprezentującej funkcję przybliżającą hipotezę.
- w – wagi sieci,
- ε – niewielka wartość będąca kryterium stopu dla algorytmu. Wykorzystuje fakt, że różnice w wartości wag będą maleć wraz ze zbliżaniem się do minimum funkcji $J(w)$.

Krok pierwszy, czyli wybranie punktu startowego jest rozwiązany przy pomocy losowej inicjalizacji wag w. Często stosuje się tu różne heurystyki dostosowane do użytej funkcji aktywacji (dla $\tanh$ jest to przykładowo inicjalizacja Xaviera).

Kłopotliwym zagadnieniem może wydawać się wyliczenie gradientu $\nabla f(w)$. Nie posiadając formalnej definicji hipotezy tylko jej aproksymację w postaci sieci neuronowej musimy posłużyć się funkcją kosztu $J(w)$ i zależnością:

gdy:

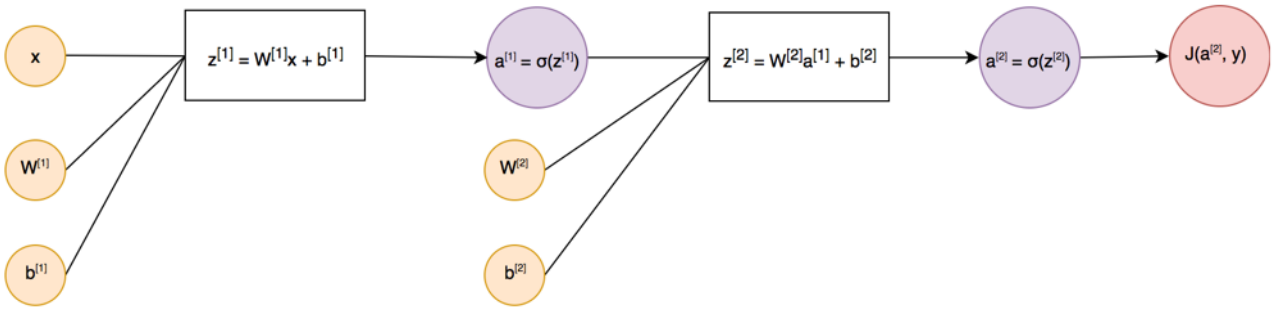
$$J(z) = J(a(z))$$

to:

$$\frac{dJ}{da} = \frac{dJ}{da} \bullet \frac{da}{dz}$$

Posługując się nimi, możemy wyjść od wzoru funkcji kosztu $J(w)$ i na tej podstawie aktualizować wartość wag ostatniej warstwy. Co więcej możemy, przy pomocy nowo wyliczonych wag wyliczyć nowe wartości wag warstwy przedostatniej i tak niejako licząc warstwa po warstwie, aktualizować wszystkie wagi sieci neuronowej.

Niestety konkretne wzory służące do aktualizacji wag muszą być wyprowadzone dla każdej z funkcji kosztu z osobna. Całe szczęście biblioteki do uczenia maszynowego mają zaimplementowane większość z nich w taki sposób, że użytkownik musi tylko zaimplementować propagację wprzód i wybrać funkcję kosztu. Nie mniej jednak, by zobrazować ten proces wyprowadzę wzory potrzebne dla propagacji wstecznej dla dwuwarstwowej sieci neuronowej. Propagację na dalsze warstwy będą tylko ponownym wykorzystaniem tych samych wzorów w zastosowaniu do kolejnych wag.



Rysunek 2.6: Przykładowy schemat sieci neuronowej, przy liczeniu propagacji wstecznej.

Dla pojedynczego przykładu ze zbioru treningowego:

$$J\left(w\right) = y \log a^{[2]} + (1-y) \log\left(1 - \ a^{[2]}\right)$$

oraz przyjmując sigmoid za funkcję aktywacji:

$$\sigma = \frac{e^z}{e^z \ - \ 1}$$

wtedy:

1. $\frac{dJ}{da^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$,
2. $\frac{da^{[2]}}{dz^{[2]}} = \frac{d\sigma}{dz}^{[2]} = \frac{e^{z^{[2]}}}{\left(1+e^{z^{[2]}}\right)^2} = \frac{1}{1+e^{-z^{[2]}}} \bullet \frac{1+\left(e^{-z^{[2]}}\right)-1}{\left(1+e^{-z^{[2]}}\right)} = \frac{1}{1+e^{-z^{[2]}}} \bullet (1 - \frac{1}{\left(1+e^{-z^{[2]}}\right)}) =$ $a^{[2]}(1-a^{[2]})$,
3. $\frac{dJ}{dz^{[2]}} = \frac{dJ}{da^{[2]}} \bullet \frac{da^{[2]}}{dz^{[2]}} = \left[-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right] \bullet a^{[2]}\left(1-a^{[2]}\right) = -\left[\frac{y\left(1-a^{[2]}\right)+a^{[2]}(1-y)}{a^{[2]}\left(1 \ \breve{} \ a^{[2]}\right)}\right] \bullet$ $a^{[2]}\left(1-a^{[2]}\right) = a^{[2]} - y.$

Dysponując $\frac{dJ}{dz^{[2]}}$, $a^{[2]}$ i $a^{[1]}$ (z fazy propagacji wprzód) możemy zaktualizować $w^{[2]}$ i $b^{[2]}$,

4. $\frac{dJ}{dw^{[2]}} = \frac{dJ}{dz^{[2]}} \bullet \frac{dz^{[2]}}{dw^{[2]}} = (a^{[2]} - y)a^{[1]}$,
5. $\frac{dJ}{db^{[2]}} = \frac{dJ}{dz^{[2]}} \bullet \frac{dz^{[2]}}{db^{[2]}} = (a^{[2]} - y)$.

Powyższe obliczenia pokazują w jaki sposób obliczyć nową wartość wag między funkcją kosztu a ostatnią warstwą sieci. Kolejne kroki pokażą w jaki sposób obliczyć nową wartość wag pomiędzy dwiema warstwami (można te kroki uogólniać na dowolną ilość warstw).

6. $\frac{dJ}{da^{[1]}} = \frac{dJ}{dz^{[2]}} \bullet \frac{dz^{[2]}}{da^{[1]}} = (a^{[2]} - y) \bullet w^{[2]}$
7. $\frac{dJ}{dz^{[1]}} = \frac{dJ}{da^{[1]}} \bullet \frac{da^{[1]}}{dz^{[1]}} = w^{[2]}(a^{[2]} - y) \bullet a^{[1]}(1 - a^{[1]})$

($\frac{da^{[1]}}{dz^{[1]}}$ obliczone analogicznie jak w kroku numer. 2),
Stąd wiadomo jak obliczyć wagi $w^{[1]}$ i $b^{[1]}$,

8. $\frac{dJ}{dw^{[1]}} = \frac{dJ}{dz^{[1]}} \bullet \frac{dz^{[1]}}{dw^{[1]}} = w^{[2]}(a^{[2]} - y) \bullet a^{[1]}(1 - a^{[1]}) \bullet x$
9. $\frac{dJ}{dw^{[1]}} = \frac{dJ}{dz^{[1]}} \bullet \frac{dz^{[1]}}{dw^{[1]}} = w^{[2]}(a^{[2]} - y) \bullet a^{[1]}(1 - a^{[1]})$

Po takim cyklu, należy wybrać kolejny przykład z zestawu treningowego, przeliczyć propagację wprzód, funkcję kosztu a następnie znów propagację w tył aż kryterium stabilności metody gradientu prostego zostanie spełnione.

3. Neuronowa sieć konwolucyjna

3.1. Wstęp

W przypadku zagadnień z dziedziny analizy obrazu, same sieci gęste robią się niepraktyczne. Teoretycznie możliwe jest przetworzenie obrazu w taki sposób, że każdy piksel traktowany jest jako jeden węzeł sieci, ale to już przy obrazkach o wymiarach 100x100x3 (100 na 100 pikseli, 3 kanały RGB) daje 30000 węzłów na warstwie pierwszej. Nawet gdyby warstwa druga miała być dwa razy mniejsza tj. 15000 węzłów macierz $w^{[1]}$ miałaby wymiar (30000, 15000) przy założeniu, że rozmiar liczby typu double to 64 bity to sama taka pojedyncza macierz ważyłaby $64\ [B] \bullet 30000 \bullet 15000 \bullet \frac{1}{8}\ [b] \bullet \frac{1}{1024}$ [kb] $\frac{1}{1024}$ [mb] $\approx\ 3422\ [mb]$. Poza problemami z pamięcią, leży wziąć pod uwagę wysoki koszt obliczeniowy operacji na tak dużych macierzach, a przecież rozdzielczość współczesnych obrazów jest wielokrotnie wyższa. By poradzić sobie z tym problemem, należało wprowadzić inny rodzaj architektury sieci neuronowej – konwolucyjną sieć neuronową.

3.2. Opis elementów składających się na CNN

3.2.1. Filtry i splot w CNN

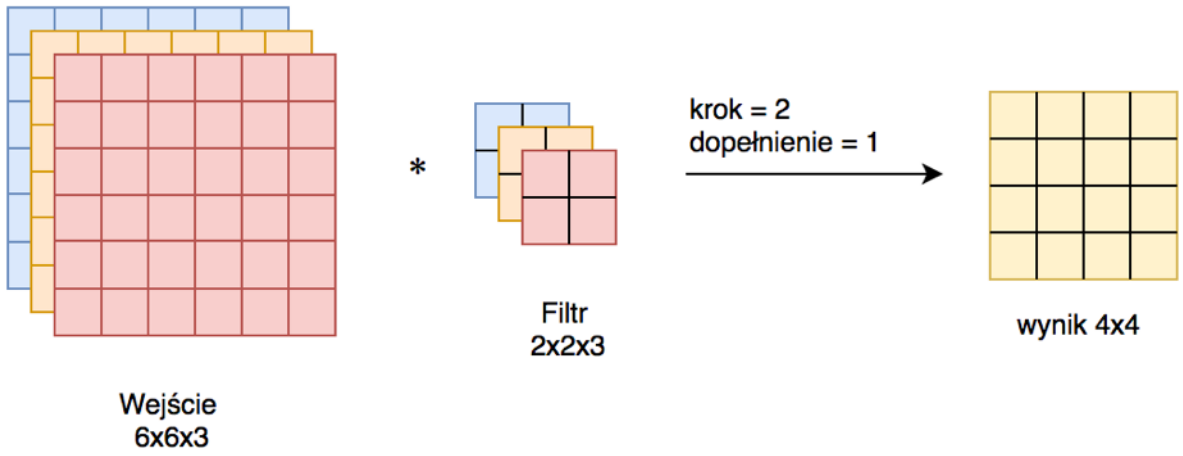
By ograniczyć ilość węzłów definiuje się tzw. filtry. Są to macierze kwadratowe wypełnione wagami w których wartość jest ustalana w procesie propagacji wstecznej. By uzyskać wynik, stosuje się na obrazie dyskretną operację splotu z zadanym *krokiem* i *dopełnieniem*.

Operację wykonaną pomiędzy pojedynczą warstwą wejściową a odpowiadającym jej filtrem można zapisać:

Niech: $k = \left\lfloor \frac{n+2p}{s} \right\rfloor$, $k\ \in Z$

$$\sum_{q=0}^{n_w} \sum_{p=0}^{k} \sum_{i=0}^{n_g-1} \sum_{j=0}^{n_g-1} \sigma[f_q\,(ps+i, ps+j) \bullet g_q\,(i,j) + b]$$

gdzie:



Rysunek 3.1: Przykładowa operacja splotu wykonywana między dwiema warstwami sieci CNN.

- n_w – wielkość trzeciego wymiaru wejściowej macierzy,
- n – wymiar wejściowej macierzy kwadratowej, w tym przypadku równy 6,
- p – dopełnienie, wymiar dodatkowej „obwódki” wokół oryginalnego obrazu w celu wyeliminowania tzw. problemu brzegu; w tym przypadku równe 1.
- s – krok, z którym dopasowujemy filtr na macierz wejściową, w tym wypadku równy 2, to znaczy macierz filtra jest aplikowana co dwa elementy w pionie i poziomie (każdy piksel jest zakryty wyłącznie raz),
- n_g- wymiar macierzy filtra,
- $f_q(i,j)$ – funkcja reprezentująca wartości macierzy wejściowej warstwy q,
- $g_q(i,j)$ – funkcja reprezentująca wartości macierzy filtra warstwy q,
- σ - funkcja aktywacji,
- b – bias.

Wzór zaproponowany powyżej jest analogicznym do gęstego modelu wzoru na propagację wprzód $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$, $a^{[l]} = \sigma(z_n^{[l]})$. Jeżeli traktować w, a i b jak macierze, a σ jako operację działającą na każdym z elementów macierzy z osobna.

Uzyskane tym sposobem macierze wynikowe oraz wagowe są wielokrotnie mniejsze, kosztem tego, że te same wagi są ustalane przy pomocy różnych części obrazu. Można za to wprowadzić wiele różnych zestawów filtrów. Każdy z nich, może nauczyć się wykrywać inne zależności w obrazie np. jeden może odpowiadać za wykrywanie krawędzi w pionie, inny w poziomie. W

tym wypadku każdy z tych filtrów miałby wymiary 2×2×3. Stosując wiele różnych filtrów na raz uzyskujemy wielowymiarowy wynik, wymiary trójwymiarowej macierzy wynikowej wynosi wtedy:

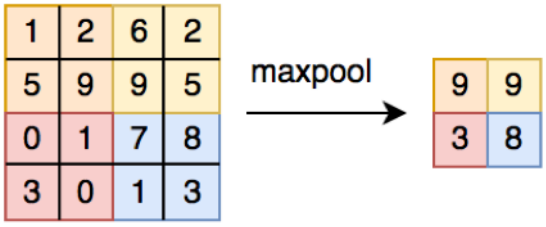
$$n_x \times n_y \times n_z$$

Gdzie $n_x = n_y = \left\lfloor \frac{n+2p-n_f}{s} \right\rfloor$, n, p, s są zgodne z opisem wyżej, n_f to wymiar macierzy kwadratowej filtra a n_z – to ilość użytych filtrów wielkości $n_f \times n_f \times n_z$.

Należy jednocześnie nadmienić, że filtry nie muszą być stosowane tylko na danych wejściowych. Wynik konwolucji może i często jest, przekazywany do głębszych warstw, gdzie uczone są filtry pracujące na wynikach warstw poprzednich. O wizualizacji tego czego się te filtry nauczyły, będzie dalej w tej pracy.

3.2.2. Warstwy łączące

Warstwy łączące (ang. "pooling layers") są często wykorzystywane w celu zmniejszania wymiarów warstwy sieci CNN. Do najpopularniejszej należy zdecydowanie tzw. max pooling. Polega na tym, że z obszaru w macierzy reprezentującej wartości danej warstwy w sieci konwolucyjnej wybieramy element z najwyższą wartością.



Rysunek 3.2: Zasada działania maxpool dla macierzy 4x4 przy użycia filtra 2x2, bez dopełnienia i z krokiem s = 2.

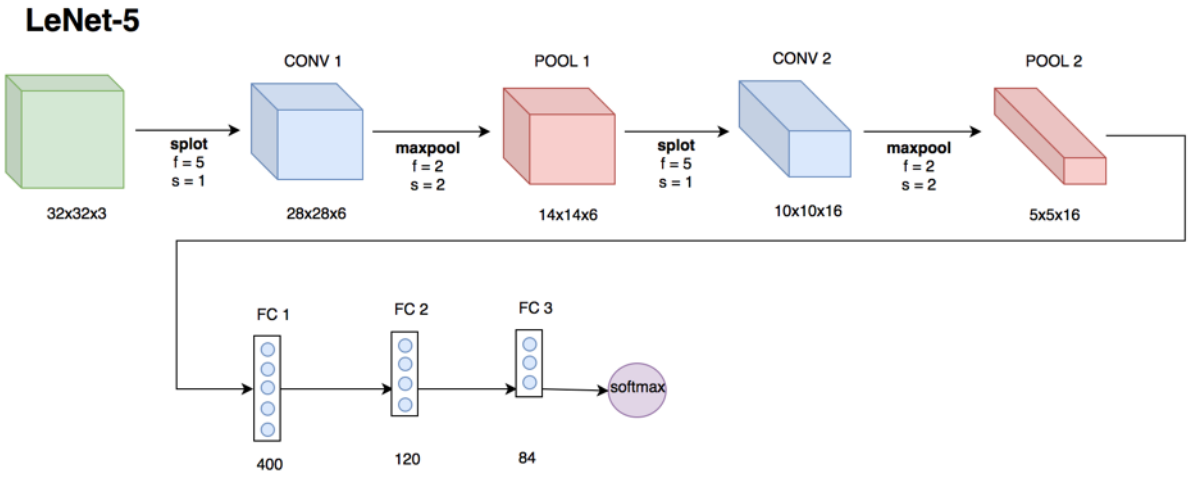
Istnieje jeszcze wariacja sumująca wszystkie wartości z obszaru (ang. „sum pooling") i licząca średnią z obszaru (ang. „avg pooling"). Zasady odnośnie liczenia wielkości macierzy wynikowej dotyczącą każdego z wariantów poolingu i są analogiczne jak w przypadku obliczania splotu, z tym wyjątkiem, że pooling stosowany jest do każdej z warstw osobno i nie wpływa na wielkość trzeciego wymiaru macierzy wynikowej.

Popularnym schematem jest ustawienia warstwy poolingowej zaraz po warstwie konwolucyjnej (wraz z wykonaniem aktywacji na wyniku). Warstwy splot-pooling często są łączone ze sobą w różnych wariantach architektury sieci CNN.

3.3. Przykład sieci konwolucyjnej - LeNet-5

3.3.1. Schemat sieci LeNet-5

Przy użyciu splotu, warstw łączących i sieci gęsto połączonych, można zaprezentować kompletny model sieci konwolucyjnej. Posłużę się przykładem klasycznej sieci nazwanej LeNet-5. Oryginalnie służyła do rozpoznawania odręcznie pisanego pisma.



Rysunek 3.3: Schemat sieci konwolucyjnej LeNet-5

Przyjmuje ona wejściu obrazki o wymiarach 32×32 pikseli z trzema kanałami (RGB), stosuje operacje splotu przy użyciu 6 filtrów o wymiarach 5×5×3 dając, zgodnie ze wzorem, wynik o wymiarach 28×28×6. Do tak uzyskanych macierzy wprowadzany jest element nieliniowości przy pomocy funkcji aktywacji, w tym przypadku tangens hiperboliczny. Następnie stosowana jest warstwa maxpool o wymiarze filtra 2×2×6. Tak uzyskany wynik, trafia na splot tym razem 16 filtrów 5×5×6 i maxpool 2×2×16. Następnie przekazywany jest do w pełni połączonej sieci neuronowej, trójwarstwowej, posiadającej 400, 120 i 84 węzły na kolejnych warstwach. W wersji, którą chce zaprezentować wynik uzyskam przy pomocy softmax i będzie to 10 elementowy wektor.

3.3.2. Implementacja sieci LeNet-5

By zaimplementować zaproponowany model, posłużę się biblioteką do głębokiego uczenia maszynowego – Keras. Model będzie zaimplementowany w języku Python (wersja 3). Sieć będzie trenowana przy pomocy CPU. Do treningu użyję publicznie dostępnej bazy danych mnist (http://yann.lecun.com/exdb/mnist/). Dane zostaną podzielone w sposób jaki został zaproponowany przez autorów tj. 60 000 przykładów będzie użytych do treningu sieci a 10 000 do ewa-

luacji sprawności sieci. Często pracuje się na trzech zbiorach danych: do uczenia, do regulacji parametrów oraz do testów. W tym wypadku architektura sieci nie będzie jednak zmieniana w trakcie treningu więc zestaw służący do regulacji zostanie pominięty.

Dane dostępne w postaci binarnej zostaną wczytane na raz do pamięci RAM. Następnie wartości pikseli obrazu zostaną znormalizowane z przedziału [0, 255] do [0, 1]. Etykiety początkowo wczytywane są jako cyfra z przedziału [0, 9], w przypadku użycia softmax, istnieje potrzeba dostosowania ich do postaci wektora składającego się z zer i jedynek w którym, jedynka wstawiona na odpowiedniej pozycji pokazuje jaka jest wartość etykiety (ang. „onehot vector").

Niestety dostępne obrazy nie pasują formatem do zaproponowanej oryginalnie sieci LeNet-5. Zamiast posiadać trzy kanały, obrazy wejściowe są czarno białe i w wymiarach 28×28 pikseli, więc filtry pierwszej warstwy konwolucyjnej będą miały odpowiednio zredukowany trzeci wymiar. By nie zmieniać zbytnio oryginalnej architektury, obrazy zostały dopełnione przy pomocy zer na brzegach do wymiaru 32×32.

Sieć trenowana jest przy pomocy wariacji metody gradientu – adam, użyta funkcja kosztu to opisana w tej pracy „cross entropy loss". Gdy sieć jest trenowana przy pomocy adam, dane użyte do treningu grupowane są w mniejsze porcje (w tym przypadku 256 elementów), na których przeliczana jest propagacja wprzód i w tył na raz. Następnie, na sieci wynikowej uczona jest następna porcja, aż do wyczerpania elementów ze zbioru uczącego. Taki jeden obieg nazywamy epochem. Podczas treningu tej sieci wykonam 5 takich epochów,

W celu uzyskania lepszej sprawności w przewidywaniach sieci, zastosowałem w implementacji, nie opisane do tej pory praktyki zapobiegające przeuczeniu sieci tzw. „dropout" i normalizację porcji (ang. „batch normalization"). Dropout, losowo „wyłącza" poszczególne neurony podczas uczenia, wymuszając równomierne rozłożenie odpowiedzialności za wykrywanie poszczególnych cech zbioru danych. Batch normalzation, natomiast skaluje wagi warstw ukrytych tak by mediana i odchylenie standardowe nie zmieniały się zbyt drastycznie. Daje to pozytywne efekty dla czasu treningu sieci i w pewnym stopniu, tak samo jak dropout, normalizuje wagi sieci.

3.3.3. Implementacja modelu przy pomocy Keras

```
#loading the dataset
train_labels, test_labels, train_images, test_images,
              train_images_orig, test_images_orig = loadTrainingData();
## model fitting below ##
X_input = Input((28, 28, 1))
# Padding to the 32x32
```

```
X = ZeroPadding2D((4, 4))(X_input)
# CONV -> BN -> RELU
X = Conv2D(6, (5, 5), strides = (1, 1), name = 'conv0')(X)
X = BatchNormalization(axis = 3, name = 'bn0')(X)
X = Activation('tanh')(X)
# MAXPOOL
X = MaxPooling2D((2, 2), name='max_pool_1', strides=2)(X)
# CONV -> BN -> RELU
X = Conv2D(16, (5, 5), strides = (1, 1), name = 'conv1')(X)
X = BatchNormalization(axis = 3, name = 'bn1')(X)
X = Dropout(0.4, name='drop_1')(X)

X = Dropout(0.3, name="drop_2")(X)
X = Dense(400, activation='tanh', name='fc_in')(X)
X = Dense(120, activation='tanh', name='fc_mid')(X)
X = Dense(84, activation='tanh', name='fc_out')(X)
X = Dense(10, activation='softmax', name='fc_softmax')(X)
# Create model
model = Model(inputs = X_input, outputs = X, name='ResNet-5')
model.compile(optimizer="Adam",
              loss="binary_crossentropy", metrics=["accuracy"])
# Fit the model
model.fit(x=train_images, y=train_labels, epochs=5, batch_size=256)
# Evaluate
preds = model.evaluate(x = test_images, y = test_labels)
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```