

Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki

Paweł Marcin Chojnacki
nr albumu: 260082
informatyka

Praca magisterska

Porównanie wydajności współczesnych architektur sieci neuronowych

Opiekun pracy dyplomowej
dr hab. Piotr Wiśniewski

Toruń 2018

Spis treści

Słownik pojęć	7
Wstęp	11
1. Neurony	15
1.1. Model perceptronu prostego	16
1.2. Co to jest sieć neuronowa?	19
1.3. Jak uczyć sieci neuronowe	20
1.4. Elementy sieci neuronowych	21
1.4.1. Metoda gradientu prostego	21
1.4.2. Funkcje aktywacji	22
1.4.3. Parametry	23
1.4.4. Hiperparametry	25
1.4.5. Wykres obliczeniowy	26
1.4.6. Wsteczna propagacja błędu	26
2. Głębokie sieci neuronowe	29
2.1. Zmiany na przestrzeni lat	29
2.2. Głębokie sieci neuronowe	30
2.3. Konwolucyjne sieci neuronowe	32
2.3.1. Współdzielenie parametrów	36
2.3.2. Warstwy gęste	37
2.3.3. Rekurencyjne sieci neuronowe	37
2.3.4. W pełni konwolucyjna sieć neuronowa	39
2.3.5. Pozostałe typy sieci neuronowych	41
3. Biblioteki implementujące uczenie głębokich sieci neuronowych	43
3.0.1. Tensorflow	43
3.0.2. Keras	46
3.0.3. PyTorch	48

3.0.4.	Tensorflow.js [Deeplearn.js]	50
3.0.5.	PaddlePaddle	52
3.0.6.	MXNet	54
3.0.7.	Caffe2	55
3.0.8.	ML.NET	57
3.0.9.	CNTK	59
4.	Architektura	61
4.1.	Znaczenie architektury dla wydajności sieci	61
4.2.	Przegląd efektywnych architektur	63
4.2.1.	LeNet	63
4.2.2.	AlexNet	65
4.2.3.	VGG Net	65
4.2.4.	GoogleNet / Inception	66
4.2.5.	ResNet	67
4.2.6.	ResNeXt	67
4.2.7.	RCNN – obrazek jest zastępczy nie ma nic w internecie interesującego (zrobić własną wizualizację)	68
4.2.8.	YOLO - You Only Look Once	68
4.2.9.	SqueezeNet	68
4.2.10.	SegNet	69
4.2.11.	Generative Adversarial Network	69
5.	Testy wydajności	73
5.1.	Dane	73
5.2.	Testowane architektury	74
5.2.1.	Środowisko obliczeniowe	75
5.3.	Proces eksperymentu	76
5.3.1.	Przegląd danych uczących	77
5.3.2.	Wyniki nauczania	79
5.4.	Optymalizacja wyników	79
5.4.1.	Wybór stałej uczenia	79
5.4.2.	Sztuczne zwiększenie ilości danych	80
5.4.3.	Cykliczny współczynnik uczenia	81
5.5.	Analiza wyników	81
5.5.1.	ResNet	82
5.5.2.	VGG	83
5.5.3.	ResNeXt	83
5.5.4.	Wide Residual Network	83

5.5.5. Inception	83
5.5.6. DenseNet	83
5.5.7. Porównanie czasu uczenia (CPU i GPU)	83
5.6. Podsumowanie	83
Podsumowanie	85
Spis rysunków	85

Słownik pojęć

Klasyfikacja binarna

Typ zadania klasyfikacji, którego wyjście stanowi jedna z dwóch wzajemnie wykluczających się klas. Prostym przykładem jest system rozpoznający wiadomości email podzielone na dwie klasy „spam” oraz „nie spam”.

Regresja logistyczna

Metoda statystyczna używana do analizy zbioru danych, w którym istnieje więcej niż jedna zmienna determinująca wyjście. Wyjście jest prawdopodobieństwem na ile obiekt wejściowy przypomina każdą z klas modelu. Regresja logistyczna jest zazwyczaj używana do klasyfikacji binarnej.

Funkcja kosztu

Funkcja służąca trenowaniu modelu regresji logistycznej. Pozwala na określenie odległości wartości wyjściowej sieci od celu (prawidłowej odpowiedzi). Celem sieci neuronowej jest zminimalizowanie funkcji kosztu dla sygnałów wejściowych nie spotkanych wcześniej.

Metoda gradientu prostego (ang. gradient descent algorithm)

Algorytm pozwalający znaleźć minimum funkcji kosztu. Technika ta minimalizuje błąd w odniesieniu do parametrów modelu zadanych do trenowania przez wyliczenie optymalnych wartości wag oraz progów.

Wykres obliczeniowy (ang. computation graph)

Dekompozycja wyrażenia (matematycznego) w pojedynczych atomowych krokach. Używany do szukania miejsc możliwego zoptymalizowania funkcji. Każdą operację wyrażenia przedstawia się jako osobny węzeł w grafie skierowanym. Wykresów obliczeniowych używa się podczas ręcznej analizy funkcji błędu oraz jako wizualizacji obliczeń w wielu bibliotekach głębokiego uczenia maszynowego.

Funkcja aktywacji

Jedna z funkcji (zazwyczaj sigmoidalna lub rektyfikowana jednostka liniowa) która przyjmuje zsumowaną wartość wszystkich wejść z poprzedniej warstwy sieci neuronowej i propaguje wartość wynikową do następnej warstwy.

Konwolucja (ang. convolution)

Termin w kontekście uczenia maszynowego odwołuje się do operacji splotu lub warstwy splotowej. Splot polega na stworzeniu macierzy będącej mieszkanką macierzy wejściowej z filtrem splotowym dając zestaw wag mniejszy od oryginalnego wejścia i tworząc mapę cech.

Łączenie (ang. pooling)

Technika polegająca na redukcji danych wejściowych z poprzedniej warstwy splotowej. W łączeniu zazwyczaj używa się maksymalną wartość lub średnią z obszaru puli, który z kolei jest macierzą kwadratową o zadanym rozmiarze (2x2, 3x3 lub 5x5). Aby zmniejszyć obraz należy ustalić rozmiar filtra oraz wielkość kroku. Następnie wykonuje się przejście po całym obrazie z nałożonym filtrem. Każde przesunięcie po sygnałach z warstwy poprzedniej zwraca nową wartość dla warstwy łączącej. Pozwala to ograniczyć ilość parametrów i wyodrębnić konkretne cechy obiektu.

Wsteczna propagacja błędu

Najważniejszy algorytm dla znajdowania optymalnych wag w sieciach neuronowych. Przechodzi on w dwie strony sieci, pierw od warstwy wejściowej do wyjściowej wyliczane są wartości każdego neuronu. Następnie wykonywany jest powrót z ostatniej warstwy do pierwszej wyliczając pochodne cząstkowe błędu w stosunku do każdego parametru (wagi i progu).

Wyrzucanie połączeń (ang. dropout)

Sposób na redukcję przeuczenia modelu względem danych treningowych przez regularyzację danych. Dokładniej polega na wyłączeniu działania losowych neuronów w trakcie uczenia, zapobiega to sytuacjom kiedy dane treningowe są rozpoznawane zbyt dobrze, a nowe sygnały nieznane wcześniej mają dużo niższą jakość rozpoznania.

Epoka

Pełne przejście przez zestaw danych treningowych, tak że każdy element został przekazany do sieci jeden raz.

Model

Reprezentacja obiektu utworzonego w wyniku uczenia sieci neuronowej. Z założenia model jest tym lepszy im dokładniej przewiduje klasę obiektu, którego wcześniej nie spotkał.

Wstęp

Celem niniejszej pracy magisterskiej jest przedstawienie nowoczesnych architektur sieci neuronowych oraz przegląd najpopularniejszych bibliotek implementujących algorytmy uczenia maszynowego, a w szczególności zoptymalizowanych do pracy z uczeniem głębokim na wielu procesorach graficznych.

Głębokie sieci neuronowe, zawierające więcej niż jedną warstwę ukrytą, zostały wynalezione we wczesnych latach 60tych XX wieku. Algorytmy uczenia maszynowego były bardzo aktywnie rozwijane w latach 1950 - 1971, fundusze na badania w czasach zimnej wojny były w USA kilkakrotnie wyższe w udziale PKB. W czasie kryzysu z roku 1981, nadzieje wojska amerykańskiego na sztuczną inteligencję wygasły powodując epokę znaną jako "drugą zimę SI" (ang. second AI winter). Brak funduszy na badania znacząco zahamował rozwój systemów samouczących i do 2012 roku była to dziedzina zarezerwowana głównie dla doktorów matematyki i informatyki (głównie statystyków). [25]

Powrót sieci neuronowych na pierwsze strony gazet popularnonaukowych spowodował Geoffrey Hinton. Wykorzystał swoje idee sprzed lat na nowym, bardzo wydajnym sprzęcie. Mowa o dwóch kartach graficznych GeForce GTX 580 3GB[1], które przez tydzień trenowały model do rozpoznania obrazów pod zawody ImageNet Competition 2012. Wcześniej zawody w rozpoznawaniu obiektów na obrazach wygrywało oprogramowanie łączące wiele algorytmów uczenia maszynowego, ale głębokie sieci neuronowe nie osiągały tak dobrych wyników, stąd nagły zwrot w kierunku narzędzia dającego tym lepsze rezultaty im więcej danych treningowych zostanie. Pierwszy raz można było zobaczyć głębokie sieci neuronowe mające trafność powyżej 80%. Po publikacji dokumentu "ImageNet Classification with Deep Convolutional Neural Networks" nastąpił gwałtowny rozwój start-upów związanych z Deep Learningiem, a firmy mające w nazwie słowa 'AI' lub 'Deep' zanotowały znaczne zwiększenie funduszy na rozwój.

Moc obliczeniowa kart graficznych i dedykowanych układów tanieje z

każdym rokiem. A zapotrzebowanie na moc obliczeniową dało przyspieszenie usługom chmurowym sprzedającym moc obliczeniową na żądanie. Zajmują się tym już nie tylko największe korporacje, ale również małe startupy jak Paperspace, Crestle, NVIDIA GPU Cloud. Kluczowe dla rozwoju dziedziny okazały karty graficzne firmy NVIDIA. Firma chcąc być liderem w rewolucji Sztucznej Inteligencji, zmieniła swój wizerunek z dostawcy rozrywki dla graczy i grafików na lidera urządzeń i oprogramowania stanowiącego podstawę systemów uczących. Przez nagłą modę, wszystkie obecnie używane narzędzia mają nie więcej niż 5 lat, oczywiście nie powstały od zera, opierają się na dużo starszych bibliotekach dla obliczeń lub narzędziach wewnętrznych wielu firm. Akademicki ruch wolnego oprogramowania pozwolił na praktykowanie dziedziny głębokiego uczenia kompletnym laikom. Przykładem może być biblioteka Caffe, napisana przez doktoranta z uniwersytetu w Berkeley. Wielu entuzjastów pomagało przez lata tworzyć wydajne implementacje algorytmów w bibliotece, którą po latach Facebook zaczął używać jako podstawowego narzędzia do przetwarzania swoich zbiorów danych. Doktor, który ją rozwijał pracuje obecnie na stanowisku dyrektora działu sztucznej inteligencji i wyznacza dalszy rozwój następcy swojego dzieła Caffe2. [26]

Dziedzina jest tak dynamiczna, że zdarza się iż literatura w czasie publikacji papierowej książki staje się nieaktualna. Dlatego wykorzystana w tej pracy literatura to głównie publikacje elektroniczne z serwisu arXiv, będące zazwyczaj jeszcze bez recenzji naukowej oraz dokumentacja narzędzi wykorzystywanych do badania wydajności.

Ilość mocy obliczeniowej potrzebnej do stworzenia dobrego modelu powoduje konieczność przedstawienia największych problemów osób praktykujących uczenie głębokich sieci neuronowych w taki sposób by uniknąć kosztownego dochodzenia do zadowalających wyników metodą prób i błędów. W rozdziale poświęconym wydajności architektur przedstawiono sposoby na uniknięcie najczęściej popełnianych błędów oraz zbiór powszechnie przyjętych praktyk dla osiągnięcia optymalnej wydajności przy zaledwie kilku próbach. Wydajność jest tu rozumiana zarówno jako czas obliczeń potrzebny do wyuczenia modelu rozpoznawania danego zagadnienia, oraz dokładność z jaką już nauczony model potrafi rozpoznać niespotkane wcześniej dane. Oba te parametry są od siebie zależne, ponieważ koszt poprawy jakości predykcji algorytmu o ostatnie 1-2% może wynosić setki tysięcy złotych, zakłada się oczekiwaną poprawność lub korzysta z gotowych modeli dodając tylko własne klasy obiektów. Na koszt składa się energia elektryczna, serwery wyposażone w karty graficzne oraz czas oczekiwania na zakończenie obliczeń.

Zmiany zachodzące w rozwoju sprzętu, techniki przenoszenia cech między modelami, tzw. transfer uczenia, pozwolą na tworzenie aplikacji w czasie rzeczywistym już za kilka do kilkunastu lat.

Architektury posiadają wiele hiperparametrów, które wpływają na precyzję modelu. Po modyfikacji wartości dowolnego należy rozpocząć pracę algorytmu od początku, dlatego dobór odpowiednich wartości na początku jest niezwykle cenny. Na rok 2017, żadna otwartoźródłowa biblioteka nie posiada wbudowanych automatów do poszukiwania optymalnych hiperparametrów. Dobieranie ich dzieje się na intuicję wyrobioną dziesiątkami prób. Istnieje kilka nowatorskich technik dobierania hiperparametrów, które ze względu na to że zostały odkryte przez mało znanych badaczy, nie są jeszcze powszechnie stosowane w dużych ośrodkach badawczych. Te techniki również są na tyle nowe, że nie ma wydawnictwa branżowego opisującego ich użycie dla praktyków. Jedyne sposoby na chwilę obecną to śledzenie publikacji naukowych wydawanych w serwisie arXiv. Kilka technik strojenia stałej uczenia w tym cykliczna zmiana stałej uczenia pozwala efektywnie zmniejszyć błąd klasyfikatora o 10-20%, co wiąże się z dużo wyższą wydajnością, przy znikomym nakładzie pracy.

Do prezentacji działania bibliotek, architektur i algorytmów użyty został gotowy zbiór danych. Zestaw danych treningowych pochodzi z serwisu Mendeley. Są to zdjęcia z prześwietleń rentgenowskich podzielone na dwie kategorie:

- zdrowe płuca,
- zapalenie płuc.

Dane są anonimowe, więc zgodne z prawem europejskim i zostały zweryfikowane przez Kaggle. Po zakończeniu pracy z danymi znajduje się porównanie z najlepszymi wynikami społeczności serwisu. [27]

Badanym obszarem jest zagadnienie klasyfikacji obrazów przy pomocy głębokich sieci neuronowych. Algorytm sieci tworzy mapę cech każdej z klas. Na przykładzie zbioru zdjęć, prezentowany jest sposób wykorzystania gotowych modeli i metody optymalizacji je pod wybraną domenę. Takie podejście pozwala uniknąć wysokich kosztów tworzenia własnych modeli od zera.

Głębokie sieci neuronowe również świetnie się sprawdzają przy zadaniach nie obejmujących zagadnień prezentowanych w niniejszej pracy. Do najczęstszych zastosowań należą systemy rozpoznawania mowy, przetwarzania języka naturalnego i systemy rekomendacji produktów. Są zagadnienia

zbyt obszerne by zostały tutaj przedstawione.

Praca została podzielona na etapy tworzenia sieci neuronowej od dołu jej struktury, po wyższe poziomy abstrakcji. Rozpoczyna od przedstawienia pojedynczych elementów i struktury matematycznej sieci. Następnie opisany jest każdy element tworzący gotową architekturę. Kolejnym elementem są już gotowe architektury i biblioteki implementujące różne rodzaje algorytmów, ze wspomaganiami na programowanie rozproszone na wiele urządzeń wyposażonych w jednostki graficzne.

Wszystkie przedstawione kody źródłowe zostały napisane w języku Python 3.6 (z wyjątkiem fragmentu porównawczego w JavaScript). Python staje on dominującym językiem w uczeniu maszynowym, kosztem najpopularniejszego na chwilę obecną języka R. [29] Python składnią jest zbliżony do pseudokodu, dodatkowo wszystkie popularne biblioteki udostępniają interfejs programistyczny dla Pythona. Książki wykorzystane w tworzeniu tego tekstu, zawierają wyłącznie kod w języku skryptowym Python (z kilkoma nawiązaniem do języków R i C++).

Rozdział 1.

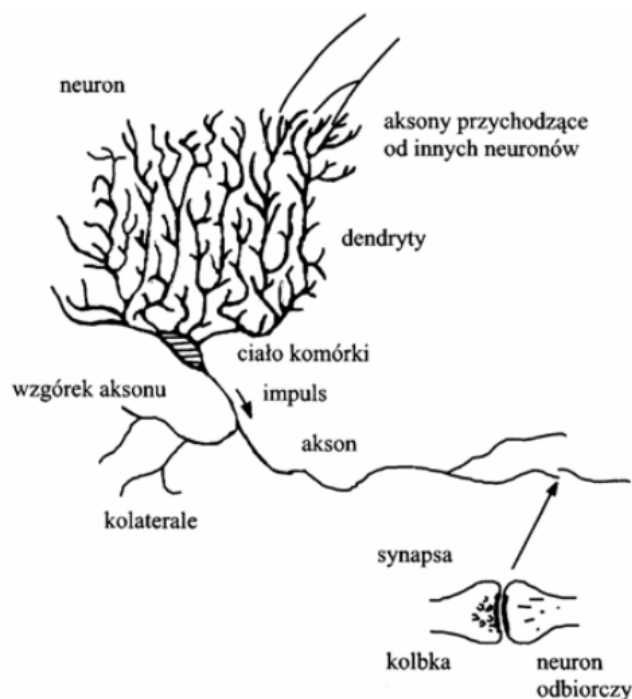
Neurony

W niniejszym rozdziale została przedstawiona idea sztucznego neuronu McCullocha i Pittsa. Opracowany model biologicznego neuronu z 1943 roku jest tak uniwersalny, że do dziś stanowi podstawę budowy sieci neuronowych. Przedstawiono tu również w jaki sposób znaleźć mapę cech, czy mówiąc prościej jak nauczyć neurony rozpoznawać wzorce. Na końcu rozdziału opisane są możliwości jakie daje łączenie sztucznych neuronów w sztuczną sieć neuronową.

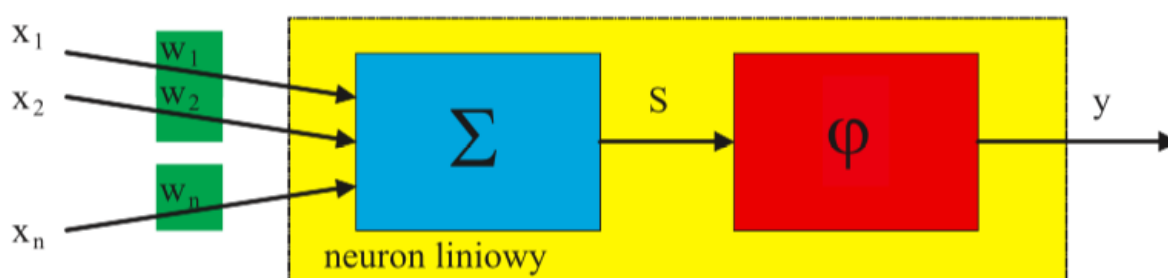
Pierwszy model sztucznego neuronu był wyobrażeniem uproszczonej biologicznej reprezentacji neuronu mózgowego z lat 40tych XX wieku. Model ten stanowił bardziej próbę implementacji idei, niż odwzorowanie prawdziwego neuronu, który jest wielokrotnie bardziej złożony, a jego zachowanie nie jest jeszcze w pełni znane. Zdefiniowany przez noblistę Ramon y Cajal'a w 1906 roku neuron jest specjalistycznym nośnikiem wszelkich informacji, a także elementem przetwarzającym wszystkie doznania i emocje, oraz sterownikiem całego ciała.

Uproszczeniem które pozwala na sprawną implementację, jest wyodrębnienie zasady przetwarzania informacji bioelektrycznej do kilku prostszych operacji. Takie uproszczenie składa się z wielu sygnałów wejściowych, wagi przypisanej każdemu z tych sygnałów, oraz pojedynczej wartości wyjściowej. Jak wielkie jest to uproszczenie widać przy porównaniu rysunku prawdziwego neuronu (również w uproszczeniu) oraz sztucznego neuronu. Taka budowa choć bardzo banalna, daje wiele możliwości. Podstawową zaletą jest prostota odwzorowania w urządzeniach elektronicznych. Przed nadejściem ery komputerów osobistych, stosowano maszyny nazywane perceptronami. Nazwa perceptron jest obecnie używana zamiennie z neuronem. [19]

Model obliczeniowy został stworzony na podstawie algorytmu logiki prostej. Nazwa neuronu sztucznego została przyjęta w kręgach akademickich



Rysunek 1.1: Wizualizacja uproszczonego modelu neuronu mózgowego.



Rysunek 1.2: Wizualizacja prostego sztucznego neuronu.

i biznesowych, stąd często osoby nie znające budowy mózgu ani działania sieci neuronowych mylnie nazywają zagadnienie Sztuczną Inteligencją, upraszczając skomplikowane procesy biologiczne do wzmacniania lub osłabiania sygnałów przekazywanych przez synapsy. Sieci neuronowe naśladują tylko jeden rodzaj pamięci, pamięć deklaratywną i w odosobnieniu nie tworzy inteligencji.

1.1. Model perceptronu prostego

Podstawowym elementem z których buduje się sieci neuronowe są sztuczne neurony, nazywane również perceptronami. Mają one być w rzeczywistości

bardzo uproszczonym odwzorowaniem komórek nerwowych występujących w mózgu. Takie uproszczenia pozwalają na łatwą implementację modelu matematycznego, który ma reprezentować nasz obiekt i być tani w replikacji. Nawet po takim uproszczeniu jest on w stanie skutecznie naśladować "uczenie". Sztuczny neuron jest funkcją matematyczną

$$f(x, w) \rightarrow y$$

. Można go opisać za pomocą modelu złożonego z:

- określonej liczby wejść $n \in \mathbb{N}$,
- wagi, skojarzonej z każdym z wejść $w_i \in \mathbb{R}, i = 1..n$,
- wybranej funkcji aktywacji $\phi : \mathbb{R} \rightarrow \mathbb{R}$

Charakterystycznym elementem budulca sieci jest wiele wejść i tylko jedno wyjście, dlatego tak łatwo stworzyć model będący funkcją matematyczną. Dane wejściowe oraz wyjściowe mogą przyjmować wartości z ograniczonego przedziału. Wartości przekazywane na wejściu i wartość wyjściowa zazwyczaj przyjmują znormalizowane wartości z przedziału $x \in [-1, 1]$ dla każdego z wejść, oraz $y \in [-1, 1]$ dla wyjścia. W uproszczeniu można przyjąć $y = \sum_{i=1}^n (w_i * x_i)$. w_i są nazywane wagami (dawniej wagami synaptycznymi) i podlegają zmianom w trakcie uczenia neuronu. Wagi stanowią zasadniczą cechę sieci neuronowych działających jako adaptacyjne systemy przetwarzania informacji. Zsumowana wartość jest wejściem dla funkcji aktywacji neuronu. Funkcja aktywacji zwyczajowo ma kształt sigmoidy, ale stosowane są obecnie również funkcje nazywane rektyfikowanymi jednostkami liniowymi. Funkcja progowa ma za zadanie symulację zachowania przekaźnika synapsy, po przekroczeniu określonego progu aktywuje się dane zachowanie lub jak na przykładzie obrazów rozpoznanie cechy.

Ta prosta jednostka stanowi dziś podstawę budowy każdej sieci neuronowej. Aby funkcja zwracała oczekiwane wyniki, wagi powinny być poprawnie ustawione. Początkowo wagi ustawiano ręcznie za pomocą operatora (osoby przełączającej fizyczne kable), który wcześniej przeliczał je dla odpowiednich parametrów wejścia

wyjścia. W latach 50tych perceptron stał się pierwszym modelem umiającym samodzielnie wyliczyć poprawnie wagi definiujące zadaną klasę na podstawie przykładów. Wagi w zależności od wartości mogą sygnał wejściowy wzmocnić gdy waga jest większa od 1, lub stłumić gdy waga jest

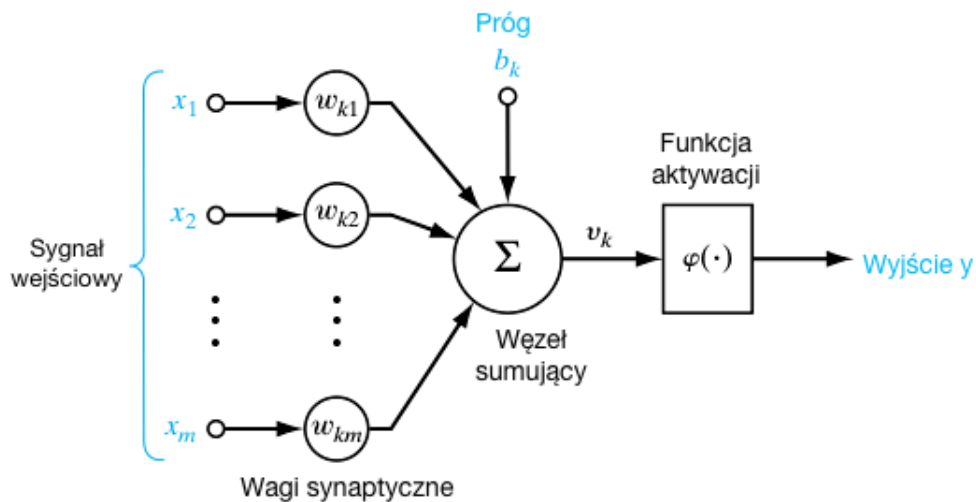
mniej niż 1. To pozwala wyuczonemu już perceptronowi na porównanie cechy obiektu wejściowego z tym co potrafi rozpoznać.

W jaki sposób sztuczny neuron jest w stanie rozpoznać sygnał wejściowy? Do wyjaśnienia zjawiska w literaturze zazwyczaj prezentuje się przedstawienie modelu w notacji wektorowej.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \quad \text{- wektor wyznaczający przestrzeń wejść, oraz}$$

$$W = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{bmatrix} \quad \text{- wektor wyznaczający przestrzeń wag.}$$

W ten notacji można wyrazić wyjście neuronu jako $y = W^T * X$. Wartość wyjściowa neuronu y , będzie wyższa im bliższe będzie położenie wektorów. [24]



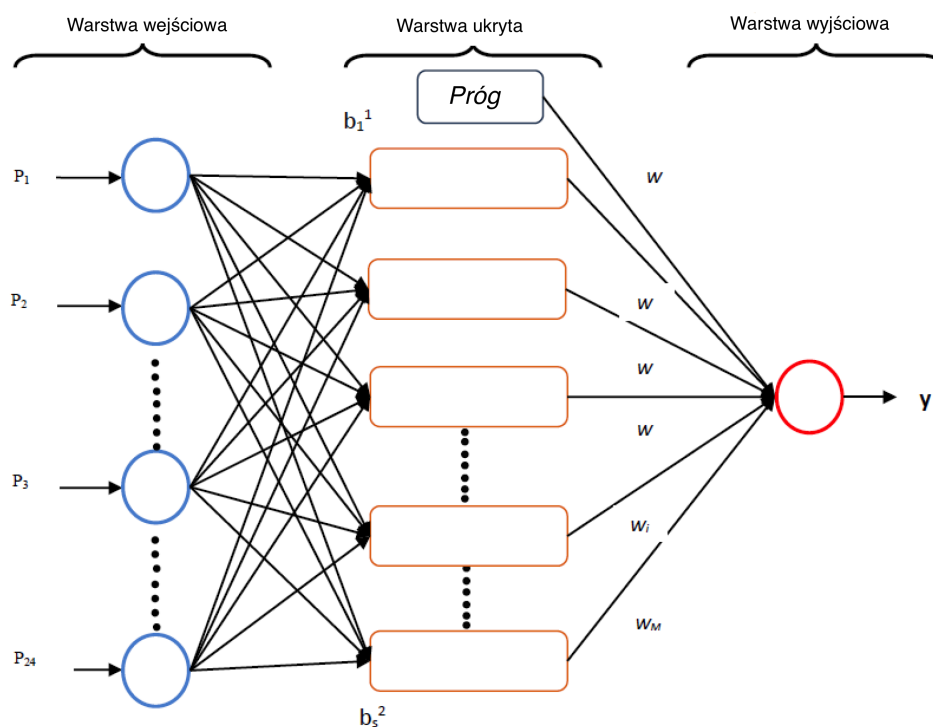
Rysunek 1.3: Prosta reprezentacja neuronu.

1.2. Co to jest sieć neuronowa?

Koncepcja sztucznej sieci neuronowej to połączenie wielu neuronów w jeden obiekt. Pozwala to na rozpoznawanie bardziej skomplikowanych wzorców i większej różnorodności typów obiektów niż klasyfikacja binarna. Zwykła (płytką) sieć neuronowa składa się z trzech warstw węzłów. Warstwa pierwsza stanowi warstwę wejściową sieci i składa się z sygnału wejściowego.

Kolejna warstwa jest nazywana warstwą ukrytą, użytkownik nie ma dostępu do niej, przypomina czarną skrzynkę. Nie można zaobserwować co się wewnątrz niej dzieje, użytkownik widzi jakie są dane wejściowe, wartości które powinny zostać zwrócone (w ostatniej warstwie), ale nie ma informacji co jest i powinno się znajdować w tej warstwie podczas trenowania sieci. Ostatnia warstwa, pojedynczy węzeł nazywany jest warstwą wyjściową. Pobiera wektor wartości wyjściowych z poprzedniej warstwy (ukrytej) i na nim zostaje obliczona wartość wyjściowa (odpowiedź).

Taka architektura nazywa się siecią dwuwarstwową. Dane wejściowe, mimo że tworzą pierwszą warstwę nie są liczone w nazewnictwie. Z warstwami ukrytą i wyjściową powiązane są parametry w (ang. weight) oraz b (ang. bias), oznaczające kolejno: macierz wag oraz wektor progów.



Rysunek 1.4: Prosta sieć neuronowa złożona z dwóch warstw.

1.3. Jak uczyć sieci neuronowe

W obecnym momencie istnieją dwie możliwości by sieć posiadała umiejętność poprawnej klasyfikacji. Można wyznaczyć dla algorytmu zbiór opisanych sygnałów wejściowych wraz z oczekiwanym sygnałem wyjściowym, ten typ tworzenia modelu sieci jest uczeniem nadzorowanym, co jest szczegółowo omawiane w tej pracy. Drugim podejściem jest podanie sygnałów wejściowych bez opisów i oczekiwanych wartości wyjściowych, to podejście nazywa się uczeniem nienadzorowanym.

Uczenie nadzorowane

Jest to typ uczenia maszynowego, które zakłada obecność ludzkiego nauczyciela. Nauczyciel zobowiązany jest stworzyć odpowiednie dane uczące. Takie dane są parą danych, wejściowego obiektu uczącego oraz prawidłową odpowiedź wyjściową do tej danej. System na podstawie tych danych ma nauczyć się przewidywać poprawną odpowiedź dla nowych danych, ze znanej mu domeny.

Zadania uczenie nadzorowanego dzielą się na dwie kategorie, regresję i klasyfikację.

W problemie regresji próbuje się przewidzieć wyniki, które są wartościami ciągłymi, czyli mając jakieś dane próbuje się je mapować na funkcję ciągłą.

W problemie klasyfikacji algorytm ma za zadanie przewidzieć wyniki będące wartościami dyskretnymi, uściślając jest to znajdowanie klas obiektów, na podstawie danych wejściowych. [39]

Uczenie nadzorowane ma wiele zastosowań, do głównych należą (wraz z używanym typem sieci):

- przewidywanie cen nieruchomości (zwykła sieć neuronowa),
- reklamy internetowe (zwykła sieć neuronowa),
- rozpoznawanie mowy (rekurencyjna sieć neuronowa),
- tłumaczenie maszynowe w translatorach (rekurencyjna sieć neuronowa),
- samochody autonomiczne (sieci hybrydowe lub inne niestandardowe sieci),
- rozpoznawanie obiektów na obrazach (konwolucyjna sieć neuronowa).

Ostatnie z wyżej wymienionych zastosowań jest tematem badanym w tej pracy.

Uczenie nadzorowane dzieli się również binarnie ze względu na strukturę dostarczonych danych. Pierwszy rodzaj danych, dane zawierające konkretną strukturę, zazwyczaj tabela, są bardzo dobrze obsługiwane przez większość znanych wcześniej algorytmów uczenia maszynowego i nie wymagają ogromnej mocy obliczeniowej. Choć sieci neuronowe świetnie się sprawdzają przy tego typu danych, używanie ich ma sens dopiero gdy danych jest bardzo dużo (wielkość tabel przekraczające miliony rekordów). Drugi typ danych, nieustrukturyzowane zdjęcia, pliki audio, tekst, jest znacznie trudniejszy do rozpoznania przez komputer, za to dużo bardziej naturalny dla ludzi. Dzięki głębokim sieciom neuronowym i wielkiej mocy obliczeniowej komputerów, dokładność algorytmów uczących się na tego typu danych znacząco wzrosła, z 70% do ok 95-99% dokładności zależnie od ilości danych.

Uczenie nienadzorowane

Uczenie maszynowe bez nadzoru nauczyciela jest drugim typem uczenia maszynowego polegającym na wyciąganiu wniosków z danych bez informacji zwrotnej o poprawności wniosków oraz bez określania etykiety danych. W algorytmach tego typu nie ma funkcji wyznaczającej poprawność utworzonego modelu. Najczęstszym zadaniem stawianym algorytmom uczenia nienadzorowanego jest wyznaczanie klastrów ze zbioru danych. W tej pracy temat uczenia bez nauczyciela nie jest poruszany.

1.4. Elementy sieci neuronowych

Sieci neuronowe do poprawnego działania potrzebują więcej niż tylko połączenia ze sobą neuronów. Należy dobrać odpowiednie algorytmy korygujące błędy. Aby algorytm zatrzymał się w odpowiednim momencie z osobna trzeba zdefiniować warunki stopu dla epoki, ilość epok, rozmiar danych. Każda warstwa musi posiadać określoną funkcję aktywacji. Ostatnim opisanym elementem sieci są hiperparametry, czyli właściwości sterujące algorytmem by wiedział w jaki sposób wyznaczać pozostałe parametry.

1.4.1. Metoda gradientu prostego

Algorytm stosowany do szukania minimum lokalnego płaszczyzny wyznaczonej funkcją. Jest to bardzo prosta metoda optymalizacji stosowana do

wyznaczania wag i progów. Płaszczyzna złożona z osi poziomych w i b oraz oś pionowa będąca wartością funkcji kosztu $J(w, b)$. Algorytm mając początkowo losowe wagi i progi, rozpoczyna w tym losowym miejscu sprawdza wartość funkcji kosztu, następnie wykonuje małe przesunięcie w najbardziej stromym kierunku w dół płaszczyzny. Operacja zejścia powtarzana jest do znalezienia do minimum lokalnego.

```
i = 0;
while i < x do
    w = w -  $\alpha * \frac{\partial J(w,b)}{\partial w}$ ;
    b = b -  $\alpha * \frac{\partial J(w,b)}{\partial b}$ ;
    i ++;
end
```

Algorithm 1: Schemat algorytmu

1.4.2. Funkcje aktywacji

Budując sieć neuronową należy zdefiniować funkcje aktywacji dla warstw ukrytych oraz warstwy wyjściowej. Funkcje aktywacji znane są również jako charakterystyka neuronu. Charakterystyka neuronu jest elementem, który pośredniczy między zsumowanym pobudzeniem neuronu, a jego danymi wejściowymi.

Najczęściej używaną charakterystyką była do niedawna funkcja sigmoidalna ponieważ ma wiele zalet:

- zapewnia łagodny gradient między wartościami 0 a 1,
- ma gładką i prostą do liczenia pochodną,
- jednym parametrem można dobrać kształt krzywej.

Definicja funkcji sigmoidalnej $f(x) = \frac{1}{1+e^{-x}}$. Obecnie nie jest zalecane używanie funkcji sigmoidalnej poza przypadkami na warstwie wyjściowej w przypadku klasyfikacji binarnej, kiedy należy stwierdzić prawdopodobieństwo klasy obiektu. Dużo wydajniejszym odpowiednikiem jest następna funkcja.

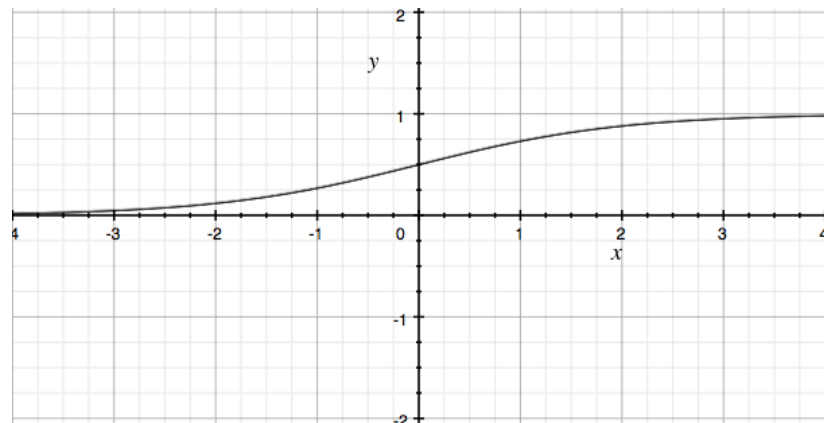
Sprawną funkcją aktywacji jest tangens hiperboliczny. Jego wartości zawierają się między -1 i 1. Definicja funkcji tangensu hiperbolicznego $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Jak widać na załączonych rysunkach ta funkcja jest przesunięciem sigmoidy względem osi y , tak że przechodzi przez punkt $(0, 0)$. Dla warstw ukrytych charakterystyka neuronu będąca tangensem hiperbolicznym ma

lepszą skuteczność. Mediana wartości wychodzących z warstw ukrytych wynosi 0. Nie trzeba wtedy stosować skalowania względem sigmoidy, co znacząco ułatwia uczenie.

Zalecaną charakterystyką neuronu w dużych architekturach jest rektyfikowana jednostka liniowa. Stała się ona domyślną funkcją stosowaną w bibliotekach implementujących sieci neuronowe. Funkcja zwraca przekształcenie nieliniowe. Na rysunku widać że funkcja bardzo przypomina liniową, składa się z dwóch liniowych fragmentów. To jest jedna z największych zalet, ponieważ funkcja ta jest banalna w implementacji i wyliczeniu wartości. Definicja funkcji jest w porównaniu do pozostałych jest banalna $relu(z) = \max(0, z)$. Jedynym problemem przy używaniu ReLU jest wyliczenie pochodnej kiedy argument z jest mniejszy od 0. W praktyce nie sprawia to większych problemów bo wartość zostanie ustawiona na 0, ale opracowano funkcję która jest pozbawiona tej niedogodności.

Nieszczelna liniowa jednostka rektyfikowana (ang. leaky Rectified Linear Unit) definiowana $f(x) = \max(x, 0.01x)$. Pozwala to zmniejszyć efekt kiedy pochyła funkcji schodzi do 0, spowalniając uczenie. Niestety ta funkcja nie jest powszechnie używana w praktyce.

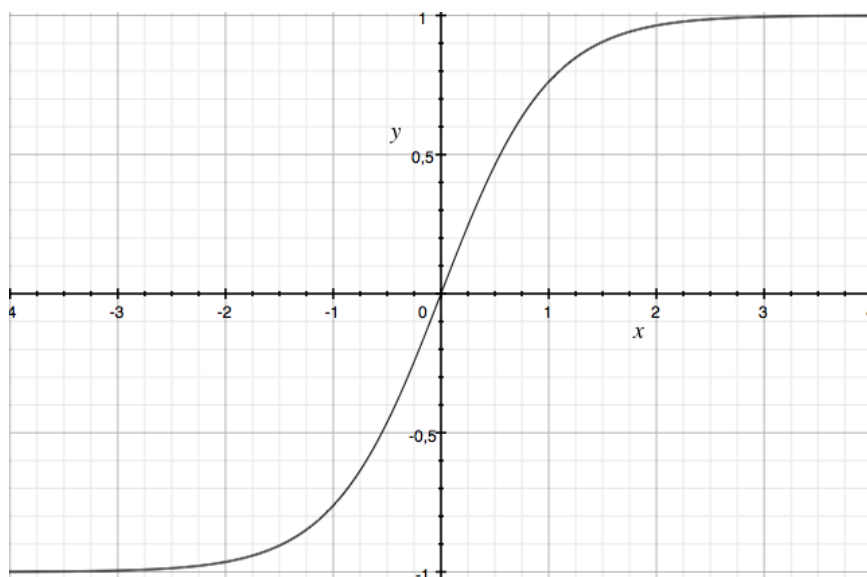
Należy podkreślić że w nowoczesnych sieciach neuronowych każda warstwa może posiadać indywidualnie ustanowioną funkcję aktywacji w zależności od architektury. Domyślnym wyborem funkcji aktywacji jest ReLU.



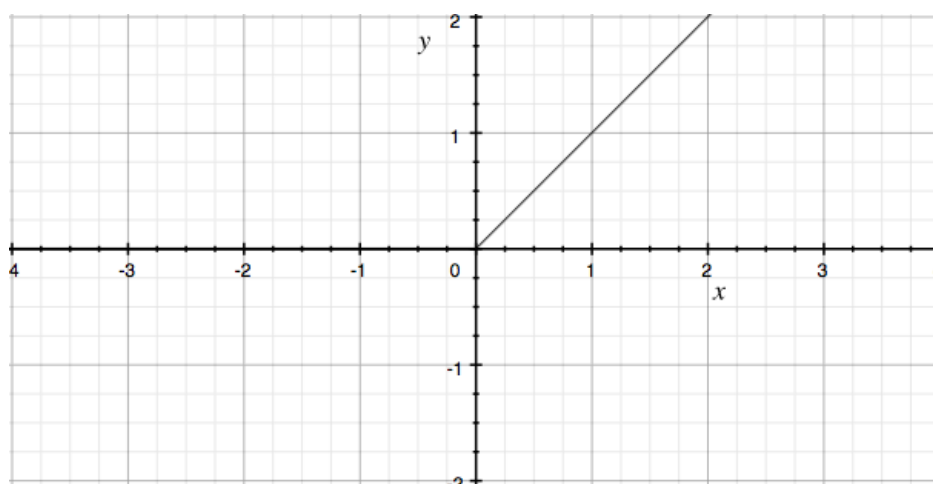
Rysunek 1.5: Wykres funkcji sigmoidalnej

1.4.3. Parametry

Najważniejszy element sieci neuronowej. Składa się na niego zbiór wag i progów (bias). Wagi wyznaczają sposób działania sieci, która ma nauczyć się dobrać odpowiednie parametry przy pomocy algorytmu uczącego, od-

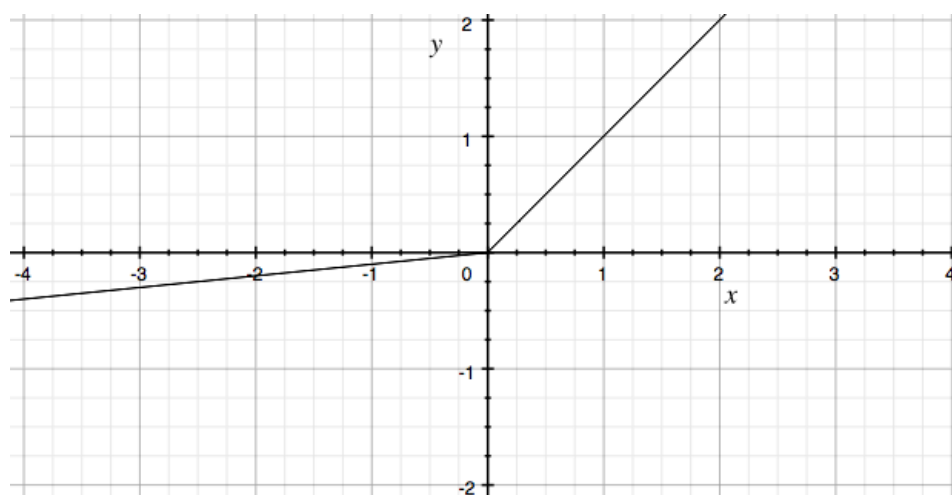


Rysunek 1.6: Wykres funkcji tangensu hiperbolicznego



Rysunek 1.7: Wykres rektyfikowanej jednostki liniowej

powiednio dobranych hiperparametrów oraz zbioru danych uczących. Wagi muszą być dobrane w sposób umożliwiający neuronom wykonanie czynności, których się od nich wymaga. Ze względu na ilość wag w głębokich sieciach, gdzie dla każdego z tysięcy neuronów może istnieć kilkaset wejść, proces musi być automatyczny. Sieć musi wiedzieć kiedy każdy z neuronów zwiększa swoją dokładność czy nie. Wagi i progi powinny być elastyczne i zmieniać się bardzo szybko na początku procesu uczenia i tylko w niewielkim stopniu zmieniać wartości kiedy algorytm kończy proces uczenia.



Rysunek 1.8: Wykres niezszerzonej rektyfikowanej jednostki liniowej

1.4.4. Hiperparametry

Tworzenie dobrej sieci neuronowej wymaga nie tylko parametrów ale również rozważnego doboru hiperparametrów. Elementy "ręcznie" dobierane dla algorytmu uczenia. Najczęściej ustawiane parametry dla sieci neuronowej to:

- stała uczenia,
- ilość iteracji uczenia w jednej epoce,
- ilość ukrytych warstw,
- ilość ukrytych jednostek,
- wybór funkcji aktywacji (ReLU, tangens, sigmoid),
- parametry regularyzacji,
- rozmiary próbek uczących,

i wiele innych mniej ważnych hiperparametrów.

Są to parametry kontrolujące sposób wykonania algorytmu uczącego i w ostateczności to one mają największy wpływ na skuteczność uczenia parametrów wag oraz progów (ang. bias). Nie ma na chwilę obecną dobrego przewodnika w jaki sposób dobierać hiperparametry. Zwyczajowo stosuje się metodę prób i błędów na podstawie tego jak algorytm się zachowuje. [4] Za każdą zmianą wartości należy uruchomić algorytm i sprawdzić jak zachowuje się funkcja kosztu, jeśli maleje lepiej niż przy poprzednich wartościach

można stroić dalej w tym kierunku. Dopieranie odpowiednich wartości nie-stety jest procesem empirycznym i należy wyrobić odpowiednią intuicję aby jak najtrafniej dobierać parametry od początku. Wartości te też nie są stałe, zmieniają się w zależności od dziedziny badanego zagadnienia. Być może najbliższe lata przyniosą dobry i spójny przewodnik dobierania najlepszych wartości, pozwoli to znacznie zautomatyzować proces.

1.4.5. Wykres obliczeniowy

Aby opisać organizację obliczeń podsekcję o wstecznej propagacji błędu, należy sformalizować operacje obliczeń. Robi się to za pomocą wykresu obliczeniowego. Węzły w grafie reprezentują skalary, wektory, macierze i tensory. Drugim elementem w grafie są operacje czyli proste funkcje z jedną lub większą ilością zmiennych. Graf obliczeniowy przydaje się kiedy istnieje zdefiniowana funkcja bądź zmienna, którą należy zoptymalizować, w przypadku sieci neuronowych oczywistą jest funkcja kosztu.

Dla przykładu chcę obliczyć funkcję kosztu J o zmiennych a, b, c zdefiniowaną $J(a, b, c) = 3(a + bc)$. Obliczenie tej funkcji składa się z trzech kroków, pierwszym jest policzenie $b * c$.

$$i = b * c$$

Następnym krokiem jest dodanie zmiennej a .

$$k = a + i$$

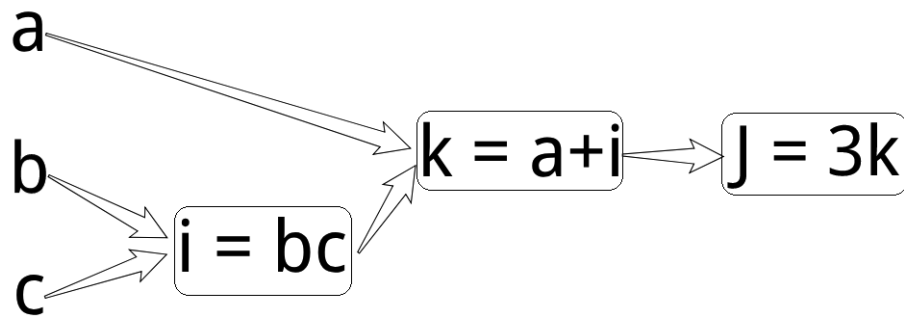
Trzecim i ostatnim krokiem jest wyjście funkcji kosztu $J = 3k$. Graf obliczeniowy powyższej funkcji znajduje się na rysunku poniżej.

Do minimalizowania funkcji kosztu algorytm oblicza pochodne w odwrotnej kolejności do przedstawionej na poprzednim przykładzie grafu.

1.4.6. Wsteczna propagacja błędu

Algorytm wstecznej propagacji błędu został opracowany w latach 70tych, jednak używany dopiero ponad 10 lat później kiedy Rumelhart, Hinton i Williams wykazali skuteczność metody w strojeniu wag w ukrytych warstwach sieci.

Przekazując sygnał na wejście sieci zostaje on propagowany w przód przez wszystkie warstwy do warstwy wyjściowej. Losowy neuron w warstwie ukrytej otrzymuje wartości wyjściowe z neuronów poprzedniej war-



Rysunek 1.9: Wykres obliczeniowy dla funkcji $J(a, b, c) = 3(a + bc)$

stwy, mnoży każdy otrzymany sygnał z wagą przypisaną danemu wejściu i przekazuje zsumowany wynik do funkcji aktywacji. Każdy neuron wykonuje takie operacje do czasu trafienia na ostatnią warstwę gdzie sieć podaje już oczekiwany wynik. Cały ten proces nazywa się propagacją w przód. Mając oczekiwany wynik na warstwie wyjściowej, zazwyczaj prawdopodobieństwo przynależności do każdej z uczonych klas należy wyliczyć funkcję błędu (ang. loss function). Funkcja ta pozwala określać odchylenie wyliczonego wyniku od właściwej odpowiedzi. Celem sieci jest zminimalizowanie błędu przez wyliczenie pochodnej z funkcji kosztu w stosunku do wag modelu.

Wsteczna propagacja błędu jest narzędziem używanym przez algorytm gradientu spadkowego aby wyliczyć gradient funkcji błędu. W trakcie przejścia wstecz przez sieć neuronową następuje korekcja wag. Przechodząc od wartości wyjściowej należy zmodyfikować wagi warstwy poprzedzającej tak by zmniejszyć wartość funkcji błędu.

Istnieją również analogiczne metod korekcji wag, jak szybka propagacja (ang. quick-propagation) oraz bardziej skomplikowane i ograniczone założeniami matematycznymi metoda gradientów sprzężonych i metoda Levenberga-Marquardta. Metody te są dużo szybsze od wstecznej propagacji błędu, ale przez swoje ograniczenia prawie nigdy nie są używane w praktyce. [19]

Rozdział 2.

Głębokie sieci neuronowe

W tym rozdziale zostały przybliżone trzy interesujące techniki głębokiego uczenia. Zostały wybrane ze względu na ich obszerną ilość zastosowań oraz potencjał, który można wykorzystać przy rozpoznawaniu wzorców na obrazach. Jedna z nich zostanie używana wielokrotnie w dalszej części pracy, jest w szczególności istotna ze względu na wykorzystanie w rozpoznawaniu obrazów. Wszystkie opisywane architektury korzystają z konwolucyjnych sieci neuronowych.

2.1. Zmiany na przestrzeni lat

Podstawowe techniki znane z głębokiego uczenia znane są już od dziesięcioleci, należy teraz odpowiedzieć na pytanie dlaczego dopiero teraz zostają wykorzystane na masową skalę przez wszystkie firmy technologiczne, a powoli także przemysłowe. Sieci neuronowe z powodzeniem były wykorzystywane do prostych zadań jak rozpoznawanie cyfr, kodów kreskowych, kodów pocztowych, pisma odręcznego od wczesnych lat 90tych ubiegłego wieku. Ze względu na ograniczenia wydajnościowe komputerów osobistych i brak odpowiedniej architektury przetwarzania obliczeń równoległych modele wyprodukowane przez sieć musiały być bardzo małe, tak małe żeby mieściły się na dysku twardym o pojemności mniejszej niż płyta DVD. Wytrenowanie modelu, który jest przechowywany na dysku, zamiast pamięci podręcznej, bądź jak obecnie w jeszcze szybszej pamięci karty graficznej, jest zadaniem tak czasochłonnym, że nie opłacało się tworzyć głębokich sieci neuronowych z tysiącami parametrów na komputerach klasy PC. Wynalezienie osobnego procesora do zadań graficznych i jego późniejsze możliwości programowania na specjalistycznym SDK umożliwiły eksperymenty z głębokim uczeniem w czasie iteracji liczonym w tygodniach, nie miesiącach. Wydajność obli-

zeniowa procesorów GPU znalazła się na odpowiednim poziomie dopiero po roku 2010. Jak opisano w pracy, jeszcze w 2012 roku wytrenowanie sieci AlexNet zajęło dwa tygodnie na dwóch kartach NVIDIA GeForce 580 GTX, najmocniejszych na tamtą chwilę na rynku dla graczy. NVIDIA na swoim blogu chwali się informacjami jakoby przyspieszenie kart graficznych między latami 2012 a 2015 wzrosło pięćdziesięciokrotnie. [40] Drugim równie ważnym aspektem nagłego rozkwitu w rozwoju głębokich sieci jest ilość dostępnych danych. Smartfony umożliwiły generowanie danych tekstowych, dźwiękowych i miliardów gigabajtów zdjęć i filmów. Taka ilość danych idealnie współgra z rosnącą wydajnością sieci neuronowych. Innowacje w zakresie algorytmów odbyły w zakresie optymalizacji szybkości obliczeń. Jednym z głównych przykładów zwiększenia wydajności jest przejście z funkcji sigmoidy jako funkcji aktywacji do funkcji ReLU. Ta funkcja liczy się dużo szybciej, ponieważ dla wszystkich dodatnich wartości, nachylenie gradientu wynosi 1, w sigmoidzie zaś gdy wartość często wynosi 0, parametry zmieniają się dużo wolniej.

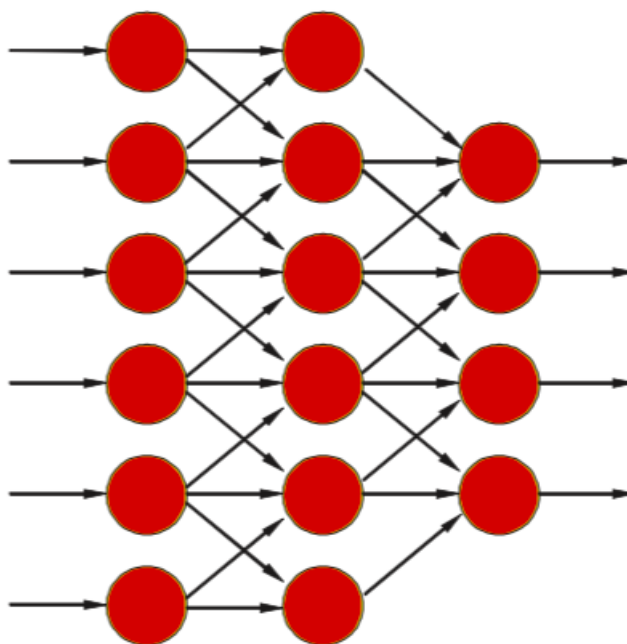
Innowacje dzieją się dużo szybciej kiedy cykl iteracyjny od pomysłu, przez kod do eksperymentu dzieje się prawie w czasie rzeczywistym, w przeciwieństwie do wcześniejszych możliwości gdy trening trwał często ponad miesiąc. Głębokie uczenie jeszcze większych sieci przyspieszy z ciągłym strumieniem nowych, opisanych danych, oraz coraz to szybszym sprzętem.

2.2. Głębokie sieci neuronowe

Elementy sztucznych sieci neuronowych zostały opisane w poprzednim rozdziale, w tej sekcji uwaga skupiona jest na podziale architektur, głębokości warstw ukrytych oraz rodzajach dodatkowych warstw, takich jak filtr cech, warstwy łączące, konwolucje.

Głębokie sieci neuronowe znane są również jako sieci neuronowe typu *feedforward* (w literaturze autor nie znalazł polskiego odpowiednika na to słowo, również prof. Tadeusiewicz posługuje się angielską nazwą), kolejna nazwa to wielowarstwowy perceptron. Nazwa **feedforward** ma naznaczyć że wszystkie informacje przepływają przez funkcję wyliczaną z x , przez pośrednie operacje wyliczające f by ostatecznie trafić do wyjścia y . Przez brak informacji zwrotnej, która mogłaby natychmiast przekazać neuronowi wyliczone wartości. W przypadku jeśli sieć neuronowa ma otrzymywać natychmiastową informację zwrotną tworzona jest rekurencyjna sieć neuronowa przedstawiana w jednej z sekcji tego rozdziału. Wszystkie wymienione okre-

ślenia opisują sieć której zadaniem jest przybliżyć funkcję f^* . Klasyfikator $y = f^*(x)$ mapuje wejście x do kategorii y . Wielowarstwowy perceptron definiuje mapowanie $y = f(x, \theta)$ ucząc się parametrów θ , w sposób pozwalający wydobyć jak najlepsze przybliżenie funkcji. Przykłady uczące serwowane sieci dają przybliżony obraz jak ma wyglądać funkcja. W typowej sieci, funkcji aktywacji jest wiele i są one wywoływane jednak przez drugą, taki łańcuch funkcji pozwala określać głębokość wybranej sieci (stąd ich nazwa "głębokie sieci neuronowe"). Dokładanie kolejnych warstw można urozmaicić przez wybieranie innego typu jednostek dla każdej z warstw, co może ulepszyć jakość architektury. O skuteczności takich rozwiązań traktuje rozdział poświęcony porównaniu efektywnych architektur. Ten rodzaj sieci jest najbardziej użyteczny do zastosowań przetwarzania obrazów i jego modyfikacje będą opisane w dalszej części pracy.



Rysunek 2.1: Przykładowa struktura sieci typu feedforward.

Każda sieć która ma więcej niż jedną warstwę ukrytą ma prawo być nazwana głęboką, jednak im sieć jest głębsza tym lepsze potrafi osiągnąć wyniki. Warstwy ukryte działają bez możliwości wglądu w nie w trakcie uczenia, programista, nie ma również kontroli nad informacjami przekazywanymi między warstwami ukrytymi.

Możliwości uczenia wielowarstwowych sieci neuronowych, sprawiają że są one idealnymi kandydatkami na zadania wymagające rozpoznawania obrazów. Używając zwykłej sieci typu feedforward, cechy obrazu zostają wyod-

Wydobywane przez ekstraktor w trakcie działania algorytmu (strojenie parametrów) a pozostałe informacje są odrzucane. Klasyfikator następnie dzieli na kategorie wektory z wyodrębnionymi cechami obrazu. Ta metoda jest skuteczna dla sieci w pełni połączonych ze sobą, jednak niesie ze sobą poważny problem uniemożliwiający stosowanie jej obecnie na większą skalę.

Zdjęcia są duże i ich rozmiar wraz z nowymi smartfonami rośnie dużo szybciej niż przepustowość kart graficznych. Taka sieć wymagałaby już tysięcy ukrytych jednostek w pierwszej warstwie ukrytej, co daje setki tysięcy lub miliony parametrów tylko na pierwszej warstwie. Mając obraz o niewielkiej rozdzielczości $250 \times 250 \times 3$, pełne połączenie warstwy wejściowej z pierwszą ukrytą warstwą wielkości 100 000 neuronów, wygeneruje 18 750 000 000 parametrów już w pierwszej warstwie. Obecne sieci z ponad czterdziestoma warstwami stałyby się zbyt wielkie na jakikolwiek komputer czy chmurę obliczeniową. Większe możliwości sieci kosztem zapotrzebowania na moc przetwarzania dostarczanych przez nie danych, pozostają poza zasięgiem większości osób i podmiotów. W pełni połączona sieć wymaga większej ilości reprezentacji (przykładów uczących) dla każdej klasy, ponieważ nie ma ona informacji o przesunięciach, różnych kątach, pochyleniu. Takie dane należałoby wygenerować automatycznie lub znaleźć wielokrotnie większy zbiór etykietowanych obrazów.

Drugą poważną wadą tej architektury jest ignorowanie ułożenia lokalnego sygnału wejściowego. Zdjęcie można w dowolny sposób poszatковать, pomieszać a wyjście pozostanie takie samo. Niestety w przypadku rozpoznania obiektu na zdjęciach jest to nie do przyjęcia. Rozwiązaniem tych problemów jest kolejny typ sieci wywodzący się z wielowarstwowego perceptronu, konwolucyjna sieć neuronowa (ang. Convolutional Neural Network).

2.3. Konwolucyjne sieci neuronowe

Wyspecjalizowany typ sieci neuronowych do przetwarzania danych ułożonych w siatkę. Sieci te są niezwykle skuteczne w przetwarzaniu siatek pikseli i nowoczesne sieci splotowe zakładają użycie wyłącznie obrazów i posiadają wbudowane optymalizacje dla tego typu sygnałów. Sieci te nie odbiegają znacząco od klasycznych feedforward'owych. Ich dodatkową cechą jest jak wynika z nazwy, liniowa operacja konwolucji (splotu) matematycznej. Wraz z konwolucją dochodzi wymagana operacja łączenia (ang. pooling). W sieciach konwolucyjnych łączy się trzy narzędzia, lokalne pola percepcji, współdzielenie wag, przestrzenne lub czasowe próbkowanie. Te narzędzia

rozwiązują również trzy problemy, przesunięcie, skalowanie i zniekształcenie obrazu.

Prosta i zwykła sieć konwolucyjna otrzymuje na wejściu obrazy o znormalizowanej wielkości i w miarę możliwości wycelowane. Każdy neuron w wybranej warstwie otrzymuje wejście ze zbioru neuronów z poprzedniej warstwy, które znajdują się w jego sąsiedztwie. Połączenia lokalne były wielokrotnie używane w modelach uczenia maszynowego. Używając lokalnych pól recepcyjnych, jednostki są w stanie wyodrębnić pojedyncze cechy obrazu takie jak krzywizny, krańce płaszczyzn. Cechy te są przekazywane w grupach do kolejnych warstw, gdzie tworzy się z nich coraz wyższe poziomy abstrakcji obiektu, by w ostatniej warstwie nadać im już konkretny kształt obiektu. Zastosowanie warstw rozpoznających najbardziej podstawowe cechy, pozwala używać jednego neuronu do rozpoznania wielu części obrazu jeśli obiektów występuje więcej. Neurony wybranej warstwy zorganizowane są na płaszczyźnie, w której wszystkie jednostki dzielą zestaw wag. Zbiór wartości wyjściowych neuronów zadanej płaszczyzny jest nazywany mapą cech. Jednostki w mapie cech wykonują tę samą operację na różnych fragmentach zdjęcia. Warstwa konwolucyjna zatem jest złożona z wielu map cech, aby zebrać wiele cech w różnych miejscach obrazu. Jednostka w mapie cech ma połączone wejścia z polem recepcyjnym o mniejszym rozmiarze.

Na końcu sieci splotowej architektura zredukuję cały obraz do pojedynczego wektora z wynikami prawdopodobieństwa klas ułożonych wgłęb.

Lokalność cech polega na zebraniu neuronów obecnej warstwy będących w sąsiedztwie ze sobą i przekazaniu ich wartości do neuronu kolejnej warstwy. Neurony oddalone o określoną odległość, nie zostają połączone do tego neuronu, a otrzymają już własne sąsiedztwo dla kolejnych neuronów wyższej warstwy. Wagi w takim układzie nazywane są filtrem. Mając wagi w_1, w_2 połączone do neuronu kolejnej warstwy, otrzyma on na wyjściu w uproszczonej formie $y = x_1 * w_1 + x_2 * w_2$, jeśli wagi w_1, w_2 będą miały wartości 1, -1 równanie zostanie uproszczone do $y = x_1 - x_2$. Prowadzi to do sytuacji w której wyjście jest największe dla wartości $(x_1, x_2) = (1, 0)$. Większa różnica oznacza przeskok wartości pikseli w obrazie, to znaczy zostaje wykryta krawędź. Operacja ta jest filtrem, ponieważ "filtruje" krawędzie obrazu. Filtr jest traktowany jak normalne wagi i jest inicjalizowany losowymi wartościami z rozkładu normalnego, a następnie w trakcie uczenia wagi filtru są jak pozostałe wagi. W literaturze można spotkać określenie "aktywacje" sieci neuronowej, odnosi się ono już do wyuczonych map cech.

Operacja konwolucji jest działaniem wykonanym na dwóch funkcjach.

W ujęciu uczenia maszynowego pierwsza funkcja jest wejściem, druga nazywana jądrem (ang. kernel) lub filtrem. Wyjście funkcji splotowej stanowi mapa cech. Jest to najintensywniejsza obliczeniowo operacja w całej sieci, dlatego należy dodawać kolejne warstwy filtrujące z rozważą. Proces splotu warstwy wejściowej z każdym jądrem polega na wyliczeniu iloczynu skalarnego między pierwszymi elementami wejścia odpowiadającym rozmiarowi filtra. Filtr jest przesuwany zgodnie z parametrem kroku (ang. stride) wzdłuż całej macierzy wejścia. Wynikowa macierz jest mapą cech wykonaną ze wszystkich możliwych pozycji splotu. Zważając iż każdy filtr jest inicjalizowany innymi wartościami początkowymi, wynikowy zestaw cech będzie reprezentował całkowicie inne atrybuty obrazu dla każdego z odpowiadającego mu filtra. Można uznać iż każde jądro tworzy własne zestawy cech, od najprostszych wzorów w początkowych warstwach sieci do złożonych obiektów w końcowych.

Dla zobrazowania kompresji jaką tworzy konwolucja warto zaznajomić się ze wzorami na wielkość warstw łączącej i filtra, zwanego też polem recepcyjnym.

$$W_2 = \left(\frac{W_1 - WF + 2 * P}{WK} \right) + 1$$
$$S_2 = \left(\frac{S_1 - SF + 2 * P}{SK} \right) + 1$$

Gdzie:

- W_2 – wysokość
- W_1 – wysokość poprzedniej warstwy
- WF – wysokość filtra
- P – margines wewnętrzny (ang. padding)
- WK – wysokość kroku
- S_2 – szerokość
- S_1 – szerokość poprzedniej warstwy
- SF – szerokość filtra
- SK – szerokość kroku

Parametr marginesu wewnętrznego (ang. padding) pozwala na sprawienie by krawędzie obrazu również brały udział w definiowaniu wyjścia sieci

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Rysunek 2.2: Wartości zerowe na krawędziach macierzy tworzą margines.

neuronowych przez objęcie krawędzi obrazu dodatkowymi zerowymi neuronami.

Operacja łączenia to nieodłączny element sieci splotowych idący w parze za operacją nałożenia filtra. Filtr z dodaniem marginesu wewnętrznego powoduje że wielkość mapy cech pozostaje taka sama jak poprzednia warstwa sygnału wejściowego, ale teraz ma dodatkowe warstwy filtru w zależności od liczby zdefiniowanej przez użytkownika. Łączenie zmniejsza wielkość mapy cech i pozwala jeszcze bardziej wyostrzyć cechy z poprzedniej warstwy oraz obniżyć ilość parametrów, a tym samym złożoność obliczeniową sieci. Dodatkowym atutem jest obniżenia ryzyka przeuczenia sieci, ponieważ największe szczegóły obrazu zostają tutaj wyeliminowane. Zostawiane są tylko cechy ogólne dla danego obiektu. Do łączenia używa się operacji $\max()$. Filtr tej warstwy o niewielkim rozmiarze 2×2 i przesunięciu o 2, wybiera maksymalną wartość z czterech pól, które są aktualnie przeglądane. W ten sposób następuje redukcja wysokości i szerokości mapy cech, głębokość (ilość filtrów) pozostaje bez zmian. Przesunięcie i rozmiar filtra są hiperparametrami, zadanie odpowiedniego dobrania ich wartości należy do zadań użytkownika sieci. Zwyczajowo nie używa się zerowych marginesów dla warstw łączących. Warstwa łącząca będzie rozmiarów:

$$W = \frac{W_1 - F}{P} + 1$$

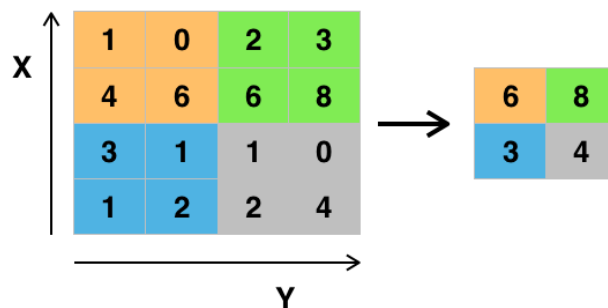
$$S = \frac{S_1 - F}{P} + 1$$

$$G = G_1$$

Gdzie:

- W – wysokość

- W_1 – wysokość poprzedniej warstwy
- S – szerokość
- S_1 – szerokość poprzedniej warstwy
- F – rozmiar przestrzeni filtra
- P – przesunięcie
- G – głębokość
- G_1 – głębokość poprzedniej warstwy



Rysunek 2.3: Operacja łączenia MaxPooling.

2.3.1. Współdzielenie parametrów

Wartości wag są współdzielone między zestaw neuronów podpiętych do wybranego jądra. Jest to technika, która umożliwia istnienie sieci konwolucyjnych. Bez tego głębokie sieci neuronowe byłyby zbyt przeładowane parametrami by podołać złożoności obliczeniowej. Założenie współdzielenia parametrów między neuronami wynika z założenia, że jeśli dana cecha, np. krawędź, występuje w jednej części obrazu, prawdopodobnie będzie się powtarzać w również innych fragmentach. Można wykorzystać zjawisko powtarzalności wzorców i używać raz zdefiniowaną cechę, na całym obrazie zamiast definiować ją na nowo dla każdego fragmentu obrazu.

Współdzielone wagi filtra oznaczają że jeden filtr jest stały dla całej powierzchni obrazu wejściowego, przesunięcie filtra po całym obrazie nie zmienia jego wag, dzieje się to dopiero podczas aktualizacji wag w trakcie wykonania wstecznej propagacji błędu. Powtarzalne cechy są znajdowane niezależnie od tego, który region obrazu wejściowego jest rozpatrywany.

Dla przykładu obraz o rozmiarach 3x3, mając nałożony filtr o rozmiarach 2x2 otrzyma do przeliczenia operacje:

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$F = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$$

$$\beta = \begin{bmatrix} w_{11}, & w_{12}, & w_{21}, & w_{22} \end{bmatrix}$$

$$F * X = \begin{bmatrix} \beta \cdot [x_{11}, x_{12}, x_{21}, x_{22}] & \beta \cdot [x_{12}, x_{13}, x_{22}, x_{23}] \\ \beta \cdot [x_{21}, x_{22}, x_{31}, x_{32}] & \beta \cdot [x_{22}, x_{23}, x_{32}, x_{33}] \end{bmatrix}$$

2.3.2. Warstwy gęste

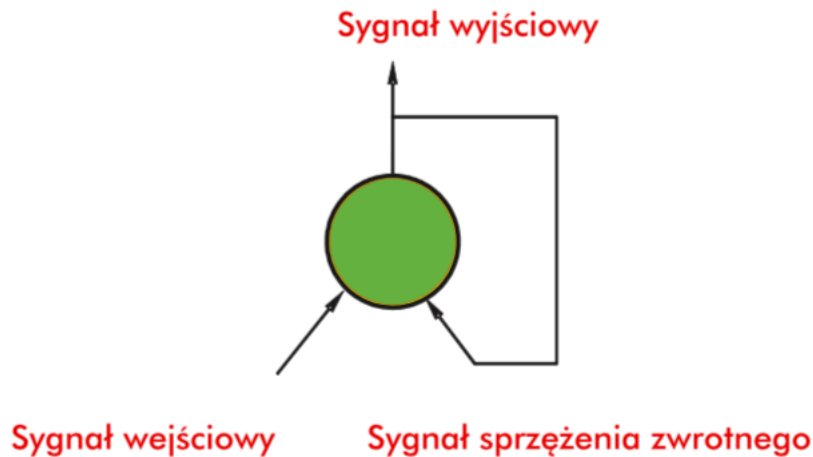
Warstwa w której wszystkie neurony są połączone ze wszystkimi neuronami kolejnej warstwy nazywa się w pełni połączoną (ang. fully connected layer) oraz warstwą gęstą (ang. dense layer). Jest to typ warstwy używany w tradycyjnych sieciach neuronowych gdzie naturalnymi są połączenia każdy z każdym. W splotowej sieci neuronowej jest to ostatni niezbędny budulec zaraz za warstwami konwolucji i warstwami łączącymi.

Warstwy gęste stosuje się na końcu architektury sieci, gdzie neurony już głosują jaka jest poprawna odpowiedź. Podczas trenowania sieci, korekcja wag sprawia że wybór klasy wyjściowej ma się odbyć na podstawie cech znalezionych w poprzednich warstwach. Kiedy już sieć jest wyuczona i otrzyma przykład do sklasyfikowania, następuje głosowanie przez wszystkie neurony, algorytm liczy średnie prawdopodobieństwo dla każdej wybranej przez każdy neuron klasy wyjściowej i wybiera jako odpowiedź o największym prawdopodobieństwie (największej średniej sumie wag). W ten sposób lista cech z poprzednich warstw staje się listą "głosów".

2.3.3. Rekurencyjne sieci neuronowe

Sieci które zawierają sprzężenia zwrotne, czyli połączenia które zwracają sygnał z neuronów do wcześniejszych warstw sieci. Ta charakterystyka daje

sieci nowe możliwości. Sieć ze sprzężeniem zwrotnym generuje zjawiska niedostępne w sieciach jednokierunkowych. Sieć jednym wejściem może wygenerować sekwencję sygnałów, ponieważ sygnały z wyjścia jednego kroku, wracając na wejście neuronów warstw wcześniejszych generują nowe, zupełnie inne wartości. Kilka ciekawych zjawisk opisywanych w sieciach rekurencyjnych to nagły wzrost i osłabienie wag sygnałów, chaotyczne błędzenie wartości po sieci. Wymienione charakterystyki są ciekawe, ale powodują małą popularność sieci rekurencyjnych przez trudność analizy zachowań i złożoność obliczeniową. Na chwilę obecną, stosowane są głównie w miejscach mających dostęp do ogromnych centrów obliczeniowych, przykładami są, asystentka Siri w urządzeniach firmy Apple oraz Google Translator. W zastosowaniach medycznych, obiecujące są zastosowania do analizy składu białek i analiza sekwencji DNA.



Rysunek 2.4: Pojedynczy neuron ze sprzężeniem zwrotnym.

Dla poprawnego zakończenia przejścia przez sieć, sygnał musi osiągnąć stan równowagi. Stan równowagi osiąga się przez mnożenie iloczynu sygnału wyjściowego z wagą wejścia. Wynik powinien być równy sygnałowi sprzężenia zwrotnego dający wartość do wytworzenia tego sygnału wejściowego.

Obiecującą charakterystyką rekurencyjnych sieci neuronowych jest posiadanie wewnętrznej pamięci. Daje to możliwość zapamiętania informacji o sygnałach wejściowych, w wielu architekturach używa się pamięci do przewidywania co jaki będzie następny sygnał. Obecnie częstotliwość zastosowań tego typu sieci może wzrosnąć jeśli zostaną zaimplementowane z uwzględnieniem problemów wydajnościowych, tzn. ograniczenie głębokości sieci i skrócenie ilości przejść sygnału przez sieć. Testując prymitywną sieć rekurencyjną pierwszym zaskoczeniem staje się ilość przejść sygnału zanim

wartości osiągną stan równowagi.

Wielu badaczy [49] przewiduje, że po rozwiązaniu problemów wydajnościowych sprzężenie zwrotne okaże się preferowanym sposobem na uczenie neuronów. W szczególności świetnie sprawdzają się dla danych sekwencyjnych, takich jak mowa, text, audio oraz obrazy. Posiadają to czego brakuje w wielu innych sieciach, kontekst. Do standardowej rekurencyjnej sieci neuronowej posiadającej tylko pamięć krótkotrwałą, można dołączyć wynaleziony w 1997 roku, mechanizm LSTM (ang. Long-Short Term Memory).

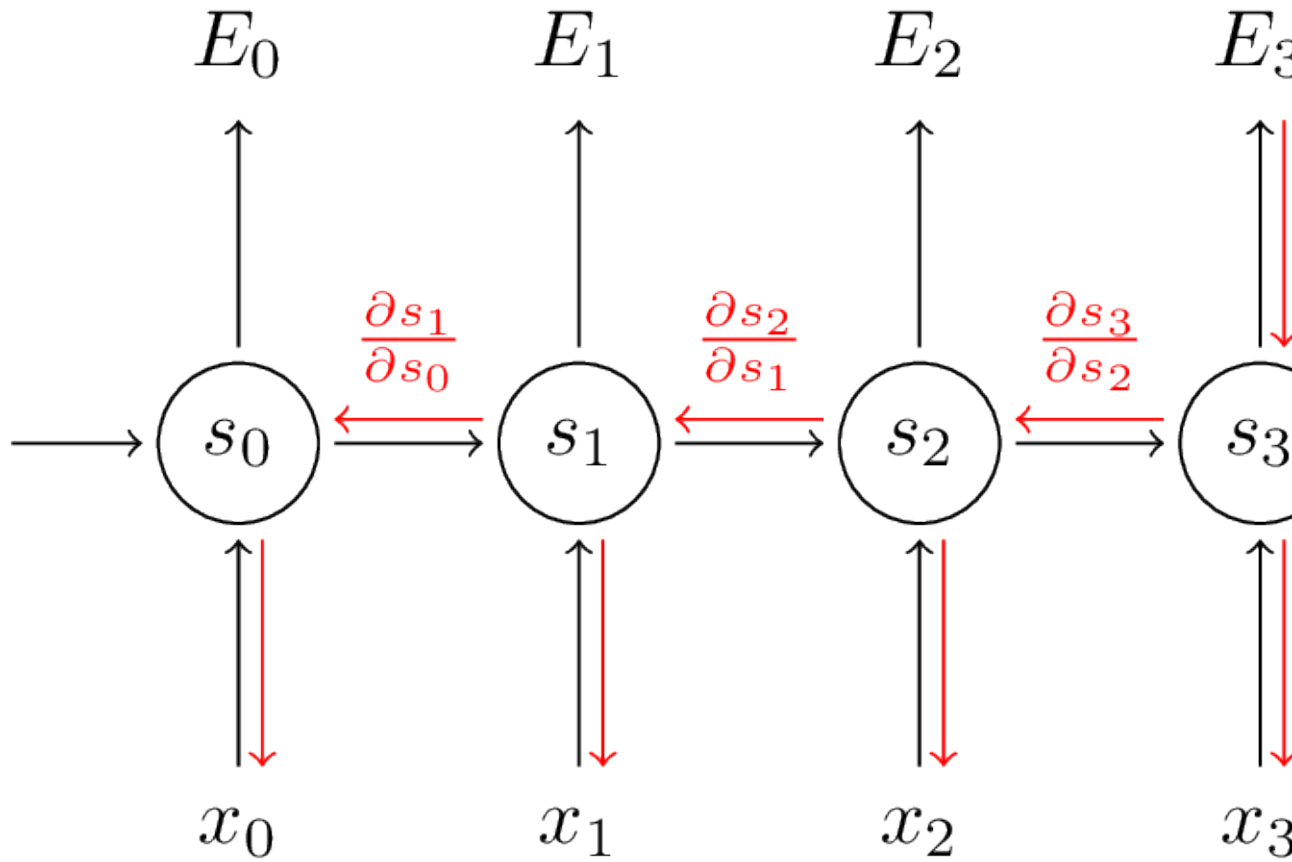
Mając dwa sygnały wejściowe sieć rekurencyjna stosuje liczenie wag na obu wejściach. Wagi są korygowane algorytmem gradientu spadkowego oraz wstecznej propagacji błędu w czasie (ang. Backpropagation Through Time). Istotną obserwacją jest możliwość mapowania jednego wejścia na wiele wyjść, wielu wejść do wielu wyjść oraz wiele wejść do jednego wyjścia, w dowolnej kombinacji ilości. Wsteczna propagacja błędu w czasie działa jak zwykła propagacja, z drobną różnicą, że sieć jest rozwijana. Korekta wag następuje od końca przez wszystkie cykle w których wystąpiła rekurencja, to znaczy jeśli sygnał przechodził przez ten sam neuron t razy, wtedy następuje rozwinięcie sekwencji dla t operacji i algorytm wykonuje korekcje jakby były to osobne sieci neuronowe zależne od siebie. Tutaj pojawia się wspomniana wcześniej złożoność obliczeniowa.

2.3.4. W pełni konwolucyjna sieć neuronowa

W budowie sieć ta przypomina normalną konwolucyjną sieć neuronową, w której ostatnia warstwa jest zastąpiona warstwą konwolucyjną z dużym polem recepcyjnym. Warstwa ta ma pomóc określić kontekst całego obrazu, np. sytuację na drodze w systemie autopilota gdzie istotne jest położenie konkretnych elementów na drodze. Bardzo ważnym jest dobranie odpowiednio dużej ostatniej warstwy konwolucyjnej, pozwoli to na dokładne określenie rozmieszczenia wykrytych obiektów. Ta warstwa jest używana jako klasyfikator każdego piksela na obrazie.

Docelowo wynik takiej sieci jest obrazem o identycznym rozmiarze co sygnał wejściowy. Obraz przypomina sygnał wejściowy w dużym uproszczeniu gdzie wykryte obiekty są zamalowane jednolicie każdy segment innym kolorem.

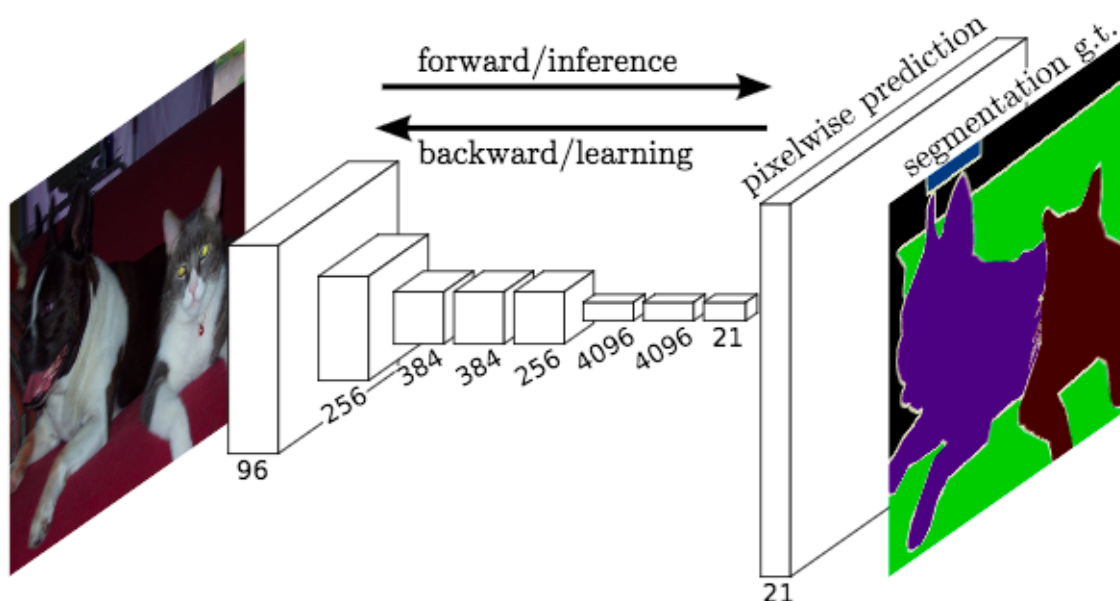
Zamiana zwykłej CNN na FCN (ang. Fully Convolutional Network) może być stosowana dla dowolnej architektury, mając przewagę nad innymi, ponieważ wykorzystanie gotowych architektur znacząco skraca czas na tworzenie działających aplikacji. Mając wytrenowaną sieć konwolucyjną, należy



Rysunek 2.5: Wsteczna propagacja błędu w czasie. x - sygnał wejściowy, s - funkcja aktywacji, E - funkcja błędu

podmienić wszystkie w pełni połączone warstwy (ang. fully connected) na warstwy konwolucyjne. Ostatnią warstwę wyjściową docelowo należy ustalić na rozmiar identyczny z warstwą wejściową z ilością kanałów odpowiadającą ilości klas.

Jak wiadomo sieci splotowe zmniejszają przestrzeń obrazów na kolejnych warstwach, więc by uzyskać znów pierwotny rozmiar obrazu na wyjściu sieci, należy skorzystać z mechanizmu konwolucji wstecznej. Jest to ta sama operacja co konwolucja, operacje są wykonywane w odwróconej kolejności, a wagi korygowane są identycznie.[46]



Rysunek 2.6: Wizualizacja architektury FCN przeznaczonej do segmentacji obrazu.

2.3.5. Pozostałe typy sieci neuronowych

Opisanie topologii pozostałych typów sieci neuronowych jest zadaniem wykraczającym za niniejszą pracę. Poniżej znajduje się lista najczęściej stosowanych topologii. Lista została opublikowana w 2016 roku przez Fjodor van Veen wraz z opisem każdego elementu.[47]

- Radial basis network,
- Long / Short Term Memory,
- Gated Recurrent Unit,
- Auto Encoder,
- Variational Auto Encoder,
- Denoising Auto Encoder,
- Sparse Auto Encoder,
- Markov Chain,
- Hopfield Network,

- Boltzman Machine,
- Restricted Boltzman Machine,
- Deep Belief Network,
- Deconvolutional Network,
- Deep Convolutional Inverse Graphics Network,
- Generative Adversarial Network,
- Liquid State Machine,
- Extreme Learning Machine,
- Echo State Network,
- Deep Residual Network,
- Kohonen Network,
- Support Vector Machine,
- Neural Turing Machine.

Rozdział 3.

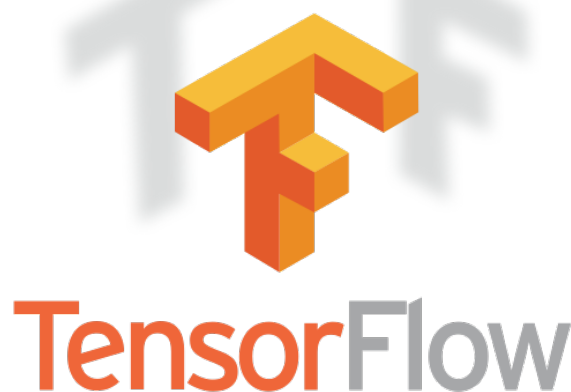
Biblioteki implementujące uczenie głębokich sieci neuronowych

Nagły wzrost zainteresowania sieciami neuronowymi, spowodował pojawienie się wielu bibliotek implementujących algorytmy do pracy z głębokimi sieciami neuronowymi, a także przy okazji wiele pobocznych zadań Machine Learningu. Obecnie wiele z największych firm technologicznych próbuje wypromować własny stos technologiczny oparty o licencje zapewniające każdemu dostęp do źródeł. Otwartość nie wynika z dobroduszości i chęci dzielenia się ze społecznością, a wspólnym mianownikiem wszystkich narzędzi jest kompletne API dla użytkowników języka Python. Jest to język w którym najszybciej można znaleźć literaturę wprowadzającą w zagadnienie. [33]

Na uwagę zasługuje fakt pojawienie się kilku bibliotek napisanych wyłącznie dla JavaScript'u. Ich zastosowanie jest nastawione na użytkowników gotowych modeli i ogranicza się głównie do importu zminiaturyzowanych wag na telefon by szybko odpowiadać na sygnały użytkownika takie jak zdjęcie z aparatu, bądź komendę głosową.

3.0.1. Tensorflow

Tensorflow jest pierwszą otwartoźródłową biblioteką dla obliczeń numerycznych używającą grafów przepływu danych. Umożliwia praktykom uczenia maszynowego wykonywać intensywne obliczenia na danych przez wydajną implementację powszechnie używanych algorytmów głębokiego uczenia. Węzły w grafie przepływu są reprezentowane jako operacje matematyczne, zaś



Rysunek 3.1: Logo biblioteki Tensorflow

wierzchołki przedstawiają wielowymiarowe macierze (tensory) zapewniające komunikację między wierzchołkami i węzłami. Architektura tworzonych sieci w Tensorflow jest łatwa w modyfikacji, dlatego jeden model można wykorzystać do pracy z procesorami CPU, GPU, urządzeniami mobilnymi i chmurą mieszaną. [18]

Pierwotnie Tensorflow umożliwiał pracę tylko z językiem Python. Od wersji 1.0 dodano eksperymentalne API do Javy i Golang. Ta wersja zadebiutowała z innowacyjnym narzędziem Tensorflow Debugger. Jest on narzędziem tekstowym do debugowania działających aplikacji.

Należy zaznaczyć że biblioteka jest wieloplatformowa i działa na wszystkich popularnych systemach operacyjnych:

- Windows
- Linux
- Mac
- Android
- iOS
- Raspberry PI

Zestaw narzędzie zawiera *Tensorflow Board* graficzne oprogramowanie do analizy utworzonych modeli. Łatwość użycia i przyjęcie modelu Open Source sprawia że jest to najczęściej używana biblioteka podczas prac naukowych, o czym świadczy ilość cytowań. [31] Wiele firm wspiera projekt używając kodu i dodając do niego własne rozszerzenia, warto wymienić choćby kilku większych kontrybutorów:

- Google,
- ARM,
- Twitter,
- Ebay,
- Intel,
- Qualcomm

Główne zalety:

- Największe zasoby dokumentacji i kursów w sieci,
- Oglądanie w czasie rzeczywistym procesu uczenia przez *Tensorboard*,
- Największa społeczność programistów skupiona wokół projektu spośród wszystkich bibliotek,
- Wspiera uczenie rozproszone na wielu maszynach,
- Biblioteki pokrewne, korzystające ze wspólnego silnika *Tensorflow Lite* i *Tensorflow.JS* umożliwiają zastosowanie tych samych modeli na urządzeniach mobilnych i przeglądarkach.

Tensorflow ma dwie wady, działające na niekorzyść osób zaczynających pracę z sieciami neuronowymi:

- Wymaga sporej ilości kodu do utworzenia działającego modelu, przez co uważany jest za bibliotekę niskopoziomową w porównaniu do pozostałych,
- Kompletne wsparcie istnieje tylko dla języka Python,
- Jest dużo wolniejszy od CNTK i MXNet [20]

Dokumentacja: https://www.tensorflow.org/api_docs/python/

Kod źródłowy: <https://github.com/tensorflow>

Licencja: Apache 2.0

3.0.2. Keras

```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(units=64, activation='relu',
                input_dim=100))
model.add(Dense(units=10, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr=0.01,
              momentum=0.9, nesterov=True))
model.fit(x_train, y_train, epochs=5, batch_size=32)
model.train_on_batch(x_batch, y_batch)
loss_and_metrics = model.evaluate(x_test, y_test,
                                  batch_size=128)
classes = model.predict(x_test, batch_size=128)
```

```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(units=64, activation='relu',
                input_dim=100))
model.add(Dense(units=10, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr=0.01,
              momentum=0.9, nesterov=True))
model.fit(x_train, y_train, epochs=5, batch_size=32)
model.train_on_batch(x_batch, y_batch)
loss_and_metrics = model.evaluate(x_test, y_test,
                                  batch_size=128)
classes = model.predict(x_test, batch_size=128)
```

Listing 3.1: Skrypt najprostszego modelu sekwencyjnego (Keras w 30 sekund)



Rysunek 3.2: Logo biblioteki Keras

Keras jest frameworkiem udostępniającym wysokopoziomowe API dla głębokich sieci neuronowych. Jego źródła zostały otwarte w 2015 roku. Społeczność skupiona wokół projektu wyróżnia się bardzo pozytywnym podejściem dla osób z poza środowiska naukowego i jest otwarta na propozycje rozwoju biblioteki. Nazwa Keras pochodzi od greckiego słowa *κερας* oznaczającego róg, geneza nazwy wywodzi się z greckiego eposu Homera pod tytułem Odyseja.

Założeniem François Chollet'a, czyli autora jest zwiększenie szybkości przeprowadzania eksperymentów, dlatego Keras ma wysokopoziomowe API

udostępniające uruchomienie algorytmów głębokiego uczenia w kilku liniach kodu. Zalety wymieniane przez autora:

- Wspieranie łatwego i szybkiego prototypowania (przez modularność, przyjazny interfejs, możliwości rozszerzenia),
- Wspiera konwolucyjne sieci neuronowe, rekurencyjne sieci neuronowe i umożliwia mieszanie obu,
- Działa na CPU i GPU.

Działanie opiera się na wykorzystaniu rdzenia Tensorflow, CNTK lub Theano jako bibliotek wykonujących obliczenia. Potwierdza to idee stworzenia narzędzia dla szybkiego prototypowania modeli. [34]

Do zalet można zaliczyć:

- Najszybsze prototypowanie ze wszystkich rozwiązań na rynku,
- Uproszczony interfejs zachęca do eksperymentów osoby nie związane z analizą danych,
- Wbudowane wsparcie do pracy na GPU,
- Wspiera strumieniowanie danych przez Spark,
- Działa nie tylko na kartach NVIDIA, ale też Google TPU, AMD

Przy dłuższej pracy i bardziej skomplikowanych modelach, wysokopoziomowość interfejsu programistycznego biblioteki ogranicza elastyczność w działaniu. Brak wielu możliwości kontroli sieci, powoduje że poleca się ją głównie dla nowicjuszy oraz eksperymentów przy standardowych typach sieci (konwolucyjnych, rekurencyjnych).

Dokumentacja: <https://keras.io/>

Kod źródłowy: <https://github.com/keras-team/keras>

Licencja: MIT

3.0.3. PyTorch

Oprogramowanie wywodzące się ze starej biblioteki *Torch*. Otwartoźródłowy zestaw bibliotek uczenia maszynowego dla języka Python. Stał się zestawem różnych narzędzi od kiedy do głównego repozytorium zaczęto dodawać inne mniejsze biblioteki oraz narzędzia takie jak serwowanie modeli



Rysunek 3.3: Logo biblioteki PyTorch

wzdłuż wszystkich platform. Główna biblioteka używana w Facebook'u do przetwarzania języka naturalnego.

PyTorch dostarcza dwie wysokopoziomowe funkcjonalności:

- Obliczenia na tensorach z akceleracją obliczeń na GPU,
- Głębokie sieci neuronowe zbudowane na automatycznie różniczkowalnym systemie taśmowym

Główne zalety:

- Proces modelowania jest prosty i przejrzysty dzięki architekturze opartej na programowaniu imperatywnym,
- Dynamiczne generowanie grafu sieci,
- Deklaratywna równoległość danych, czyli dzielenie sygnałów wejściowych na wiele mniejszych,
- Najbardziej "pythoniczna" składnia,
- Zawiera sporą ilość gotowych do użycia modeli i architektur, pozwalając na kombinację ich w nowe struktury,
- Graf obliczeniowy jest generowany w trakcie działania programu, co pozwala na używanie dowolnego debuggera i nawet drukowanie obecnego stanu obliczeń na ekran, jest to spory potencjał na narzędzia interaktywne,
- Służy także jako alternatywa dla NumPy z możliwością obliczeń na GPU,
- Specjalnie napisane zarządzanie pamięcią GPU, pozwalające na niesamowicie wydajne zarządzanie pamięcią w porównaniu do konkurencji.

PyTorch jest jednym z najmłodszych narzędzi, dlatego ma kilka nierozwiązanych jeszcze problemów:

-
- Nie jest zalecany do zastosowań produkcyjnych. Obecna wersja jest numerowana 0.4, wersja 1.0 była zapowiedziana na lato 2018. Na wrzesień 2018 brak informacji kiedy nastąpi premiera,
 - Brak gotowych narzędzi do monitoringu i wizualizacji procesu uczenia i podglądu grafu sieci

[35]

Język *Pyro* do programowania probabilistycznego stworzony przez firmę Uber opiera się na PyTorch.

Dokumentacja: <https://pytorch.org/docs/stable/index.html>

Kod źródłowy: <https://github.com/pytorch/pytorch>

Licencja: Brak określonej. Prawa autorów zastrzeżone, oprogramowanie "as is".

3.0.4. Tensorflow.js [Deeplearn.js]



Rysunek 3.4: Logo biblioteki Tensorflow.js

W 2017 roku został opublikowany projekt o nazwie Deeplearn.js z celem wprowadzenia uczenia maszynowego i głębokiego uczenia do przeglądarek internetowych bez konieczności korzystania z zewnętrznego API.

JavaScript kojarzony jest jeszcze często przez programistów nie aktualizujących wiedzy z programowania front-end'u jako powolny interpretowany język ograniczony do pracy w obrębie jednego procesu procesora CPU. [30] Aby uniknąć restrykcji wydajnościowych zaimplementowano wykonanie operacji na WebGL, interfejsowi OpenGL działającym na przeglądarkach internetowych.

W maju 2018 zespół odpowiedzialny za rozwój DeepLearn.js został połączony z zespołem programistów pracujących przy Tensorflow, a biblioteka zmieniła nazwę na Tensorflow.js. Dostarcza całej mocy Tensorflow do przeglądarki czy dowolnego interpretera kodu Javascript jak Node.js.

Tensorflow.js nie ma wspólnej bazy kodu z Tensorflow, również filozofia działania jest inna. Należy podkreślić że zbieżność nazw wynika z połączenia pracy dwóch zespołów w jedną jednostkę. Interfejs programistyczny udostępniany przez framework javascriptowy udostępnia zarówno niskopoziomowe elementy do uczenia maszynowego oraz wysokopoziomy interfejs wzorowany na tym z biblioteki Keras do tworzenia sieci neuronowych.

Biblioteka na podstawowym poziomie dostarcza dwa elementy:

- CoreAPI – komponent zarządzający kodem niskopoziomowym,
- LayerAPI – komponent zbudowany na CoreAPI, udostępnia abstrakcje wyższego poziomu

Podstawową jednostką danych jest tensor, czyli zbiór wartości liczbowych ułożonych w tablicę jedno lub więcej wymiarową. Tensor definiuje się konstruktorem *tf.tensor*, gdzie należy określić tablice oraz kształt tensora. Istnieje wiele metod pozwalających na stworzenie tensorów różnych kształtów takich jak *tf.zeros* do wypełnienia podanego kształtu zerami, oraz takie jak *tensor1d*, *tensor2d*, *tensor3d*, *tensor4d* dla ułatwienia czytelności przy tworzeniu najczęściej używanych rozmiarów. Tensory są niezmiennie co oznacza że raz stworzony tensor nie może zmieniać wartości.

Zmienne (ang. variables) są inicjalizowane wartościami tensorów co daje możliwość modyfikowania ich wartości, używa się ich do trzymania i aktualizacji wartości w trakcie trenowania modelu.

Operacje (ang. operations [Ops]) to metody pozwalające na modyfikacje danych. *Tensorflow.js* udostępnia całą gamę operacji do pracy z machine learningiem. Działają one na tensorach przez zwracanie wyniku operacji jako nowy tensor.

Powyższe struktury danych należą do CoreAPI, druga warstwa operacji LayerAPI operuje na wyższym poziomie abstrakcji i zapewnia zupełnie inne bloki do budowy sieci neuronowych. Najważniejsze atuty tej warstwy to wysokie podobieństwo do kodu tworzonego w Keras, możliwość naśladowania stylu programowania znanego z języka Python. Poniżej prezentowany jest kod porównawczy między pythonową wersją tego samego kodu i javascriptową reprezentacją z użyciem *Tensorflow.js*. Kod wykonujący intensywne operacje napisany jest asynchronicznie. Największą zaletą może się jednak okazać brak konieczności używania NumPy, którego w JavaScript brakuje. Wszystkie operacje matematyczne wymagane do uczenia maszynowego z NumPy zostały zaimplementowane w bibliotece. [36]

```

import * as tf from '@tensorflowjs/tfjs';
const model = tf.sequential();
model.add(tf.layers.dense({units: 1, inputShape:
    [1]}));
model.compile({optimizer: 'sgd', loss:
    'meanSquaredError'});
const xs = tf.tensor2d([[1], [2], [3], [4]], [4, 1]);
const ys = tf.tensor2d([[1], [3], [5], [7]], [4, 1]);
await model.fit(xs, ys, {epochs: 1000});
model.predict(tf.tensor2d([[5]], [1, 1])).print();

```

Listing 3.2: Proste operacje w JavaScript z Tensorflow.js

```

import keras
import numpy as np
model = keras.Sequential()
model.add(keras.layers.Dense(units=1,
    input_shape=[1]))
model.compile(optimizer='sgd',
    loss='mean_squared_error')
xs = np.array([[1], [2], [3], [4]])
ys = np.array([[1], [3], [5], [7]])
model.fit(xs, ys, epochs=1000)
print(model.predict(np.array([[5]])))

```

Listing 3.3: Prosty model w języku Python używając Keras

Dokumentacja: <https://github.com/tensorflow/tfjs>

Kod źródłowy: <https://js.tensorflow.org/>

Licencja: Licencja: Apache 2.0

3.0.5. PaddlePaddle

Pełna nazwa biblioteki ”**PA**rallel **D**istributed **D**eep **L**Earning”. Oprogramowanie skierowane głównie na jeden rynek, mało znane w Europie / USA. Jest to zaś najpopularniejszy framework w Chinach. Tworzony i utrzymywany przez firmę Baidu, znaną jako chiński klon Google. Założona w 2000 roku przeniosła pomysł na wyszukiwarkę internetową na chiński rynek i od tej pory z powodzeniem kopiuje wszystkie technologie i pomysły Google, w



Rysunek 3.5: Logo biblioteki Paddle

tym autonomiczne samochody.

Problemem jaki może napotkać osoba nie znająca języka chińskiego jest częste zgłaszanie uwag i komentarzy do kodu bez opisu w języku angielskim. Kursy praktycznego wykorzystania narzędzia udostępniane w internecie pochodzą głównie od pracowników firmy Baidu, a kierowane są najczęściej dla studentów chińskich uniwersytetów, stąd bardzo mała popularność poza rodzimym rynkiem. Ze strony domowej można wyczytać często powtarzające się słowa o "prostocie użycia", "prostocie działania".

Przechodząc do treści technicznych, PaddlePaddle opisywany jest jako zestaw narzędzi i bibliotek do uczenia głębokiego bazujący na "języku programowania do uczenia głębokich sieci neuronowych". Biblioteka jest bardzo elastyczna i pozwala na tworzenie dowolnego kształtu modeli. Jak w Tensorflow tak i tutaj istnieje graficzne narzędzie wizualizacji nauczania *Visual Deep Learning*, narzędzie pozwala wyświetlić wydajność trenowania i statystyki danych takie jak dokładność, wartości funkcji kosztu, rozkład parametrów, próbki obrazu i dźwięku, wykres ONNX modelu.

Na podstawie biblioteki istnieje cała społeczność i inicjatywa *EasyDeepLearning*, mająca ułatwić firmom nie zatrudniającym specjalistów od algorytmów uczenia maszynowego na wykonanie szybkich i sprawnych modeli bez konieczności posiadania terabajtów danych. Społeczność skupia się głównie w ośrodkach akademickich i start-upach w Chinach, gdzie PaddlePaddle jest standardem.

Same możliwości biblioteki są potężne, pozwala na budowę:

- Botów dzięki technologii Word2Vec,
- Systemów rekomendacji,
- Klasyfikatorów obrazów,
- Translacji maszynowej języków z użyciem głębokiego uczenia,

-
- Detekcji obiektów na obrazie filmów z użyciem Single Shot Multibox Detector,
 - Analizatora nastrojów

Przy bliższym spotkaniu, widać że jest to klon Tensorflow, który ze względu na brak konkurencji w Chinach jest dużo szybciej rozwijany, a jego możliwości w zakresie uczenia na farmach serwerów przewyższają pierwowzór. Z produktów opartych na Baidu korzysta prawie miliard osób, stąd bardzo duży nacisk na optymalizację i własne rozwiązania technologiczne przekładają się na wysoką wydajność.

Dokumentacja: <http://www.paddlepaddle.org/documentation/en>

Kod źródłowy: <https://github.com/PaddlePaddle/Paddle>

Licencja: Apache 2.0

3.0.6. MXNet



Rysunek 3.6: Logo biblioteki MXNet

Biblioteka do głębokiego uczenia stworzona przez fundację Apache. Wspiera największą ilość języków ze wszystkich dostępnych bibliotek:

- Python,
- C++,
- Cloujure,
- Julia,
- Perl,
- R,
- Scala

O popularności i powszechne przyjęciu świadczy implementacja na chmurach Microsoft Azure, Intel i Amazon Web Services. Główne zastosowania do których MXNet jest używany to rozpoznawanie mowy i pisma ręcznego, przetwarzanie języka naturalnego i przewidywanie zdarzeń. Użycie skupia się w kręgach przedsiębiorstw, trudno zatem znaleźć badania akademickie przeprowadzane z wykorzystaniem tej biblioteki.

Kolejnym atutem MXNet jest przenośność modeli. Są one zoptymalizowane by nie zajmować dużo pamięci, pozwala to na przeniesienie modelu wytrenowanego w chmurze do urządzenia mobilnego jak smartfon. Łatwe serwowanie modeli, skalowanie na żądanie z mieszaniem GPU i CPU jest zdecydowanie największymi atutami dlatego Amazon wybrał tę bibliotekę jako pierwszą na swoje centra obliczeniowe. [37]

Gluon jest API wysokiego poziomu dostarczany wraz z MXNet. Jego zadaniem jest dostarczyć czystej, spójnej i prostej składni bez obniżania wydajności działania aplikacji. Istnieje również druga warstwa API wysokopoziomowego (można wybierać, z której chce się korzystać) o nazwie Module. Ta warstwa skupia się na pracy z Symbol, paczką do przetwarzania wyrażeń symbolicznych. [21] Jego najlepszymi atutami są:

- uproszczenie składni budowanych sieci neuronowych, przypominają one pseudokod,
- elastyczna struktura zapewniająca kod imperatywny,
- dynamiczny graf sieci, umożliwia zmiany struktury sieci w trakcie działania programu,
- ładowanie sieci neuronowej do pamięci podręcznej poprawia wydajność, w tym celu używa się modelu sieci *HybridSequential*

Dokumentacja: <https://mxnet.apache.org/api/>

Kod źródłowy: <https://github.com/apache/incubator-mxnet>

Licencja: Apache 2.0

3.0.7. Caffe2

Biblioteka wywodzi się z architektury Caffe, narzędzia dla praktyków głębokich sieci neuronowych udostępniającego przejrzystą składnię API w czasach przed pojawieniem się bibliotek wspieranych przez wielkie korporacje.



Rysunek 3.7: Logo biblioteki Caffe2

Pierwsza wersja napisana przez Yangqing Jia w trakcie doktoratu na uniwersytecie Berkeley, obecnie jest dyrektorem działu AI w Facebook'u, gdzie rozwijany jest Caffe2. [23] O wieku świadczy dobór języków dostępnych do wykorzystania w bibliotece:

- C++
- Python
- MATLAB

Caffe2 jest rozwinięciem tego modelu, również pracuje na GPU, CPU, jest skalowalny, posiada spore wsparcie społeczności. Różnica jest w modyfikacji architektury obliczeniowej. Główną przewagą jest prosta skalowalność na wiele maszyn i urządzenia mobilne. Taki podział na komponenty i elastyczność w wykorzystaniu podzespołów nazywana jest odframeworkowaniem (ang. un-framework) poprzedniej wersji. [22]

Cała baza została napisana w języku C++ i była testowana latami przez Facebook'a do rozpoznawania obiektów na obrazie, analizy wiadomości do rekomendowania reklam, systemów rekomendujących na podstawie profilu użytkownika. Wydajność jest to jedną z największych zalet, po opracowaniu odpowiedniej architektury, cały kod wdrożeniowy na produkcję może być napisany w C++. Na jednym procesorze NVIDIA DGX-1 prędkość przetwarzania wynosi 231 obrazów na sekundę. [28]

Caffe2 posiada zoo modeli, gotowe do użycia wcześniej wyuczone modele dla najczęściej spotykanych zastosowań bez potrzeby wydawania zasobów na tworzenie modelu od zera. Pełna lista modeli, z którymi można eksperymentować jest pokaźna (na dzień obecny największa ze wszystkich dostępnych zwierząt w ZOO):

- AlexNet,
- GoogleNet,
- RCNN ILSVRC13,
- Densenet 121,
- Detectron,
- Inception v1,v2,v3,
- Resnet 50,
- SqueezeNet,
- VGG 19,
- ZFNET 512

Dojrzałość produktu sprawiła że istnieje wiele narzędzi programistycznych ułatwiających pracę z biblioteką. Programiści aplikacji mobilnych mają do dyspozycji wtyczki integracyjne do Visual Studio, Android Studio, Xamarin. Dla pozostałych Caffe2 jest dostępne na każdy popularny system operacyjny oraz u dostawców usług chmurowych. Kod napisany na jedną platformę jest przenaszalny na pozostałe bez konieczności modyfikacji.

Od kwietnia 2018 roku, repozytorium Caffe2 jest przenoszone do katalogu PyTorch. Facebook zdecydował się na zwiększenie kompatybilności przez stopniowe połączenie bazy kodu. Docelowo Caffe2 ma być głównym modulem dla rozwiązań mobilnych, a PyTorch dla całej reszty.

Dokumentacja: <https://caffe2.ai/docs/>

Kod źródłowy: <https://github.com/caffe2/caffe2>

Licencja: Apache 2.0 (Caffe 1 – BSD)

3.0.8. ML.NET

Nowe oprogramowanie od Microsoftu, data wydania pierwszej wersji pogładowej 7 maj 2018. Microsoft tworząc to oprogramowanie wybrał jako grupę docelową programistów skupionych w ekosystemie .NET. API biblioteki udostępnione jest w językach C++ oraz C#, przy czym kompletna dokumentacja jest tylko dla języka C#. Docelowo ma stać się częścią .NET Core. Oprogramowanie od początku istnienia powstaje jako otwartoźródłowe, a Microsoft przy każdym wydaniu zachęca do zgłaszania uwag oraz



Rysunek 3.8: Logo biblioteki ML.NET

propozycji dalszego rozwoju lub nowych funkcjonalności. Jest to odejście od tradycyjnego modelu utrzymywania oprogramowania w tajemnicy do czasu premiery. Nastawienie na łatwość integracji z oprogramowaniem biznesowym, chmurą Azure oraz łatwość obsługi przez programistów pracujących w technologiach Microsoftu ma zachęcić do wykorzystania oprogramowania na szerszą skalę kiedy wyjdzie ono z fazy rozwojowej i zastąpi CNTK.

W odróżnieniu od pozostałych na chwilę obecną ML.NET jest biblioteką głównie przeznaczoną do uczenia maszynowego, wyewoluowaną z wewnętrznych rozwiązań Microsoftu od lat używanych w zespołach Windows, Bing, Azure. Microsoft przyznaje że jest to przepisywanie większego systemu do świata otwartego oprogramowania. [32] Możliwości biblioteki można rozszerzać przez podłączenie modeli z innych bibliotek, wspierane od początku są Tensorflow, Caffe2, CNTK oraz Accord.NET. Sporym atutem jest natywne wsparcie w Azure każdej nowej wersji natychmiast po premierze. Wersja 0.3 przyniosła wsparcie dla formatu modeli ONNX. Narzędzia do głębokiego uczenia pojawiły się w wersji 0.5, zaprezentowanej 12 września 2018. Integracja modeli z Tensorflow umożliwia przeniesienie środowiska z testowego do modelu E2E (ang. Enterprise-2-Enterprise) "Przedsiębiorstwo-do-przedsiębiorstwa". Jest to jeszcze zbyt niedojrzała biblioteka by opisywać jej architekturę, nie jest nawet zalecana do rozwiązań produkcyjnych, programiści .NET prawdopodobnie będą z niej chętniej korzystali po pojawieniu się pierwszej kompletnej wersji, co ma nastąpić do wydarzenia *BUILD 2019*. Obecnie traktowana jest jako sposób na przyciągnięcie do ekosystemu .NET Core osób tworzących rozwiązania dla przedsiębiorstw. ML.NET ma jeszcze wiele braków i wad (konieczność programowania skalowania, brak wsparcia platform mobilnych i wyłączność na chmurę Azure), które uniemożliwiają

traktowanie go jako poważnego narzędzia komercyjnego w przeciwieństwie do kolejnego narzędzia Microsoftu CNTK.

Dokumentacja: <https://docs.microsoft.com/en-us/dotnet/machine-learning/>

Kod źródłowy: <https://github.com/dotnet/machinelearning>

Licencja: MIT

3.0.9. CNTK



Rysunek 3.9: Logo biblioteki CNTK

Nazwa kompletna "Microsoft Cognitive Toolkit". Jest to zestaw narzędzi stworzonych w dziale badań i rozwoju Microsoftu, zajmującego się Deep Learningiem. W 2015 zgodnie z nowym modelem biznesowym, nastawionym na częściowe otwieranie kodu źródłowego narzędzi używanych wewnątrz Microsoftu, CNTK został opublikowany na platformie GitHub w kwietniu tego roku. Microsoft wcześniej korzystał z powodzeniem z biblioteki do trenowania modeli usług krytycznych biznesowo. Architektura jest skupiona wokół głębokiego uczenia sieci neuronowych, a dopiero później dodano pozostałe popularne algorytmy uczenia maszynowego.

CNTK posiada zaimplementowane najczęściej używane narzędzia do głębokich sieci neuronowych, takie jak struktury sieci konwolucyjnych czy rekurencyjne sieci neuronowe. Także algorytmy spadku gradientowego i wsteczna propagacja błędów zostały zaimplementowane z automatycznym wykonywaniem równoległych obliczeń z pojedynczego CPU na wiele procesorów GPU na wielu komputerach w sieci. Biblioteka ta jest jednym z backendów do biblioteki Keras, jednak w przeciwieństwie do Keras'a działa tylko na

systemach Windows i wybranych dystrybucjach GNU Linux.

Niesamowicie wartościowym atutem jest z pewnością dokumentacja dostarczana przez Microsoft. Nie tylko jest ona przejrzysta i spójna z resztą dokumentacji ekosystemu .NET, ale również dostarcza dziesiątek tutoriali krok po kroku jak budować zaawansowane modele, czyścić dane i wyciągać z nich wnioski. Przystawia nawet w jaki sposób przygotowane już rozwiązania przenieść na chmurę Azure i umożliwić korzystanie z nich klientom. Dla osób nie mających podłoża w Data Science, są gotowe przepisy, czyli przygotowane wcześniej szablony do działania na Azure z interfejsem graficznym instruującym jak przygotować dane aby otrzymać gotową usługę. [38]

Dokumentacja: <https://www.microsoft.com/en-us/cognitive-toolkit/>

Kod źródłowy: <https://github.com/Microsoft/CNTK>

Licencja: MIT

Rozdział 4.

Architektura

4.1. Znaczenie architektury dla wydajności sieci

Elementy sieci neuronowych można ułożyć na nieskończenie wiele sposobów, wystarczy dodać kolejną warstwę neuronów lub zmodyfikować wielkość warstwy i wyniki będą zgoła inne. Celem szukania odpowiedniej dla danego zadania architektury jest otrzymanie najlepszej wydajności klasyfikacji przy skończonym i możliwie jak najkrótszym czasie uczenia. Chcąc uzyskać dobre rezultaty bez eksperymentów od zera, należy wybrać architekturę, która została wykorzystana do rozwiązania podobnego zadania bądź przetestować kilka gotowych architektur z wyuczonymi modelami, a następnie sprawdzić jak się zachowują dla posiadanych zbiorów danych na różnych hiperparametrach.

Architektury ze względu na bardzo zróżnicowaną wydajność obliczeniową zyskały niesamowicie na znaczeniu. Stworzenie efektywnej architektury przesądziło o wynikach zawodów "ImageNet Competition 2012". Pierwsze miejsce w klasyfikacji obrazów wyłącznie z użyciem głębokiej sieci neuronowej zapoczątkowało prawdziwą rewolucję nie tylko wśród społeczności uczenia maszynowego, ale całego przemysłu technologicznego. Pierwszy zespół, któremu udało się zejść z błędem rozpoznania do mniej niż 25% Geoffrey Hinton, zwany ojcem chrzestnym głębokich sieci neuronowych (ang. Godfather of Deep Learning) udowodnił że jego przekonania, którymi się kierował w trakcie kilkudziesięcioletniej kariery naukowej są słuszne. Poprzeczką, której profesor Hinton nie mógł wcześniej przeskoczyć były ograniczenia obliczeniowe procesorów, kiedy technologia GPU wraz z możliwością programowania równoległego w CUDA umożliwiły przetestowanie idei, podejście do rozpoznawania obrazów zmieniło świat przemysłu, a wkrótce zmieni życie każdego człowieka. Dlatego Gartner, uważa że już w 2019 roku

głębokie uczenie maszynowe stanie się kluczowym elementem w skutecznym wykrywaniu oszustw i prognozowaniu awarii.[48]

Rok 2017 został uznany za czas kiedy wysokie wyniki na zawodach "ImageNet Competition" oznaczały rozwiązanie problemu klasyfikacji obrazu. Błędne rozpoznanie algorytmów wśród najlepszych drużyn wyniosło zaledwie 2%. Przez pięć lat od czasu debiutu *AlexNet* powstało wiele architektur sieci neuronowych, bibliotek wyspecjalizowanych w szybkim tworzeniu nowych prototypów, a te dające obiecujące wyniki zostały podzielone względem zastosowania. Obecnie wyróżniane typy sieci neuronowych są sklasyfikowane względem typu danych wejściowych. Dla sygnału wejściowego w postaci pikseli najlepiej sprawdza się konwolucyjna sieć neuronowa. Warstwy konwolucyjne potrafią bardzo dokładnie wyodrębnić cechy obrazu na różnych poziomach złożoności, od pojedynczych kresek do kształtów całych obiektów, jak kontur ręki czy marka samochodu. W rozdziale drugim zostało wyjaśnione działanie tych sieci.

Polskim ekspertem i popularyzatorem sieci neuronowych, od wielu lat jest profesor doktor habilitowany inżynier Ryszard Tadeusiewicz. Osoba bardzo zasłużona w naukach technicznych, trzykrotny rektor Akademii Górniczo-Hutniczej, dwukrotnie odznaczony orderami Odrodzenia Polski, a także zdobywca tytułu Mistrza Mowy Polskiej. Jego książki stały się częścią bibliografii niniejszej pracy. Profesor w swoich publikacjach istotnie zwraca uwagę na architekturę jako element konieczny w budowaniu wydajnych systemów uczących.

Architektury różnią się ilością warstw ukrytych, typem neuronów i funkcji aktywacji w każdej warstwie. "Właśnie taka [warstwowa] struktura sieci wyjątkowo łatwo i wygodnie da się wytwarzać zarówno w formie modelu elektronicznego, jak i da się symulować w formie programu komputerowego. Dlatego badacze przyjęli właśnie strukturę warstwową i od tej pory stosują ją we wszystkich sztucznych sieciach neuronowych. [...] W związku z tym wszyscy tak postępują, nie martwiąc się ani przesłankami biologicznymi, ani dowodami wskazującymi, że architektura sieci bardziej wymyślnie dostosowanej do charakteru zadania może znacznie lepiej realizować stawiane zadania."

Tak profesor Tadeusiewicz pisał w 2007 roku, kiedy jeszcze złożone głębokie sieci neuronowe nie były wykorzystywane ze względu na koszt uczenia. Największe architektury do tej pory składały się z 5 warstw ukrytych, a zbiory danych na których pracowano były stosunkowo niewielkie w porównaniu z dzisiejszymi zasobami skatalogowanych obrazów. Prof. Tadeusiewicz

uważa że, struktura warstwowa, gdzie każdy neuron jest połączony z każdym z warstwy kolejnej to najlepsze ułożenie neuronów ze względu na prostotę. Biolodzy zajmujący się strukturą mózgu znaleźli podobne struktury (neurony ułożone warstwami) w niektórych częściach ludzkiego mózgu (w tym w oku, które jest ewolucyjnie złożone z tej samej tkanki co mózgowa).

Najbardziej wydajna obliczeniowo jest sieć z połączeniami gdzie neurony są połączone tylko z wybranymi neuronami kolejnej warstwy. Takie rozwiązanie jest fizycznie niemożliwe do ułożenia przy sieciach złożonych z milionów parametrów. To samo zachowanie jest symulowane przez zerowanie części połączeń kiedy sieć się uczy. Efekt jest ten sam, to rozwiązanie wymaga trochę więcej mocy obliczeniowej, by sieć mogła rozpoznać idealny rozkład połączeń.

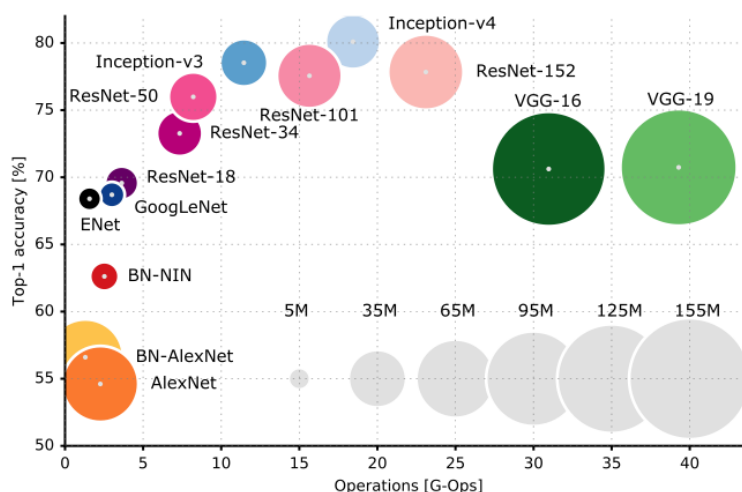
Różne modyfikacje architektur warstwowych pojawiają się pod wymogi zadań coraz bardziej dokładnego procesu uczenia. Przewiduje się że architektura będzie coraz bardziej złożona wraz ze wzrostem sprzętu. Opracowanie procesu nauczania przez przenoszenie wzorców rozpoznawania z jednej domeny do innej, daje drogę do stworzenia niesamowicie złożonych sieci. Ponieważ raz stworzony model, niezależnie od kosztu uczenia, będzie mógł być wykorzystywany do szybkiego rozpoznawania nowych zadań przed nim stawianych.

Dowodem na znaczenie ułożenia neuronów w strukturze warstwowej o różnych typach warstw jest ich rosnąca skuteczność. Od kiedy badacze eksperymentują z różnymi kombinacjami, co roku pojawiają się coraz wydajniejsze struktury. Coroczne zawody *ImageNet*, w których udowadnia się że struktura sieci ma znaczenie są świetnym przykładem że jest to proces iteracyjny i jak podkreślano wcześniej, metoda empiryczna jest obecnie jedynym sposobem na znalezienie idealnego modelu.

4.2. Przegląd efektywnych architektur

4.2.1. LeNet

[Yann LeCun, Leon Bottou, Yoshua Bengio, Patrick Haffner, 1998] - Pierwsze praktyczne zastosowanie sieci konwolucyjnej do rozpoznawania cyfr. Sieć oparta o tę architekturę używana była do czytania kodów pocztowych, oraz krótkich ciągów liczb. Systemów implementujących LeNet używało około 10% banków i instytucji pocztowych w Stanach Zjednoczonych Ameryki. Architektura została zaprezentowana w publikacji "Gradient-Based Lear-



Rysunek 4.1: Wydajność architektury względem ilości operacji potrzebnych do przejścia sygnału przez nią.

ning Applied to Document Recognition”.

Głównym przesłaniem publikacji jest pokazanie że systemy uczące lepiej nadają się do systemów rozpoznawania wzorców niż te polegające na opisywaniu wszystkich reguł. Używając najnowszych na te czasy osiągnięć uczenia maszynowego, autorzy prezentują w jaki sposób zastąpić ręcznie wykonane ekstraktory cech używając dobrze zaprojektowanego agenta, który pracuje wyłącznie w oparciu o piksele. Opierając się na paradygmacie programistycznym sieci transformatorowej, pozwalającej wszystkim modułom optymalizować kryteria wydajności dla całościowego zbioru domeny, wykluczono wszystkie reguły opisujące zbiór danych do rozpoznania.

Do czasu publikacji sieci LeNet, systemy rozpoznawania wzorców były budowane używając dwóch modułów. Pierwszy, ekstraktor cech, przekształcał sygnały wejściowe na wektory lub ciągi znaków złożone z symboli. Pozwalało to na łatwe porównywanie i łączenie elementów. Ekstraktor cech zawiera sztywne reguły, które są specyficzne dla zadania. Drugim elementem jest klasyfikator, moduł ogólnego przeznaczenia, podatny na uczenie. Główną bolączką takich systemów jest osoba projektująca zbiór cech. Jakość rozpoznawania wzorców bowiem zależy od kreatywności i ogromu pracy jaki włoży projektant pierwszego modułu w opisanie i rozpoznanie wszystkich możliwie występujących cech w zbiorze danych, na którym system ma pracować. Dodatkowo do tego podejścia zniechęca fakt, iż dla każdego innego systemu należy cechy projektować od początku. Nie ma znaczenia że dane wejściowe są bardzo podobne. Dla przykładu rozpoznawanie liter i cyfr, są tutaj rozpatrywane jako dwie zupełnie różne domeny, dla których należy

zaprojektować osobne ekstraktory cech. Rozwiązaniem tych problemów jest ujednolicenie systemu i wyrzucenie modułu opierającego swoją wydajność na żmudnej pracy projektanta.

Lecun z zespołem, zaproponowali system oparty o "gradientową propagację wsteczną". Składa się ona z 7 warstw, bez wejściowej, ale licząc z wyjściową. Pełna nazwa LeNet-5. Liczba w nazwie oznacza ilość warstw ukrytych. Sieć w porównaniu z opisywanymi w następnych sekcjach jest najmniejsza (pamięciowo) i najprostsza w implementacji. Czas wytrenowania takiej sieci na nowoczesnym GPU zawiera się w mniej niż jednej minucie. Najlepszym zbiorem uczącym dla LeNet jest MINST, zbiór cyfr pisanych odręcznie, około 70 000 obrazów. Zbiór został przestudiowany przez wielu ekspertów od wizji komputerowej i jest to jeden z najstarszych zbiorów obrazów używanych do dziś w uczeniu maszynowym. Połączenie LeNet wraz z zestawem danych uczących MINST uchodzi za klasyczny wstęp do sieci neuronowych jako najprostszy przykład. Porównuje się je często do programów "Witaj, świecie!" w językach programowania.

Sieć na wejście przyjmuje sygnał złożony z obrazów 32x32 piksele w różnych odcieniach szarości (bez kolorów). Największa cyfra w bazie ma rozmiar 20x20 pikseli wyśrodkowane na polu o rozmiarach 28x28 pikseli. Wartości danych wejściowych są znormalizowane, wartość tła (białe piksele) wyrażona jest wartością -0,1, a wartość rozpoznawanych danych (czarne piksele) odpowiada wartości 1,175. Pozwala to uzyskać średnie wartości 0 i wariancje na poziomie 1, takie wartości znacząco przyspieszają proces uczenia. Używa filtra konwolucyjnego o rozmiarach 5x5 pikseli z przesunięciem o 1.

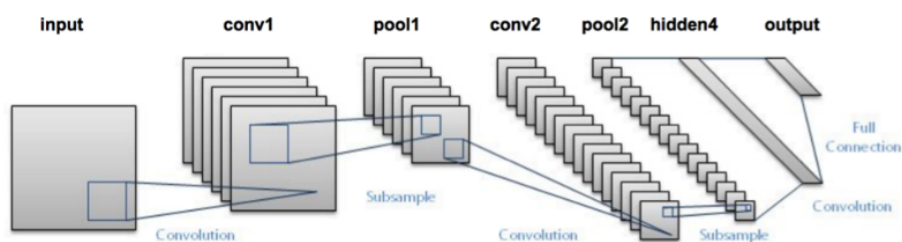
Pierwsza warstwa jest warstwą konwolucyjną w sześcioma mapami cech. Każdy neuron w mapie cech jest połączony do wejścia sąsiadującej drugiej warstwy o rozmiarach 5x5. Rozmiar każdej mapy cech wynosi 28x28 pikseli, ma to zapobiegać wyjściem poza granice obrazu. Pierwsza warstwa zawiera 156 parametrów i aż 122 304 połączenia.

Druga warstwa jest przeznaczona do próbkowania i jak tak samo jak pierwsza, posiada 6 map cech o większym rozmiarze bo 14x14 pikseli. Każdy neuron w każdej z mapy cech jest połączony w sąsiedztwie rozmiarów 2x2 z cechą poprzedniej warstwy. Cztery wejścia do neuronu w drugiej warstwie są dodane, następnie mnożone przez współczynnik wag, na koniec wynik do wyniku operacji dodawany jest próg. Wynik podawany jest jako argument dla funkcji sigmoidalnej. Pola recepcyjne nie nachodzą na siebie, dlatego mapy cech są połowę mniejsze od tych w pierwszej warstwie. Druga warstwa

ma 12 parametrów i już tylko 5 880 połączeń.

Trzecia warstwa jest warstwą konwolucyjną, tak samo jak pierwsza. Jest zaś większa od pierwszej i posiada 16 map cech. Każdy neuron w każdej mapie cech jest połączony do kilku sąsiadów o rozmiarach 5x5 pikseli w identycznych lokalizacjach podzbioru mapy cech drugiej warstwy. Nie łączy się wszystkich map cech drugiej warstwy z trzecią celem ograniczenia liczby parametrów oraz uniknięcia symetryczności sieci. Kiedy sieć nie jest symetryczna, każda mapa cech może nauczyć się zupełnie innych parametrów obrazu. Schemat połączeń działa następująco: pierwsze sześć map cech z warstwy trzeciej jest zasilanych wejściem z wyjścia kolejnych trzech map cech poprzedniej warstwy.

Wyjście pierwszej warstwy konwolucyjnej daje rozmiar 28x28 pikseli złożony z 6 map cech. Następnie przychodzi warstwa druga "łącząca" o rozmiarach 14x14 pikseli złożona z 6 map cech, każda jednostka mapy cech jest podłączona z sąsiedztwem o rozmiarach 2x2 odpowiadającym cechom z pierwszej warstwy. [7]

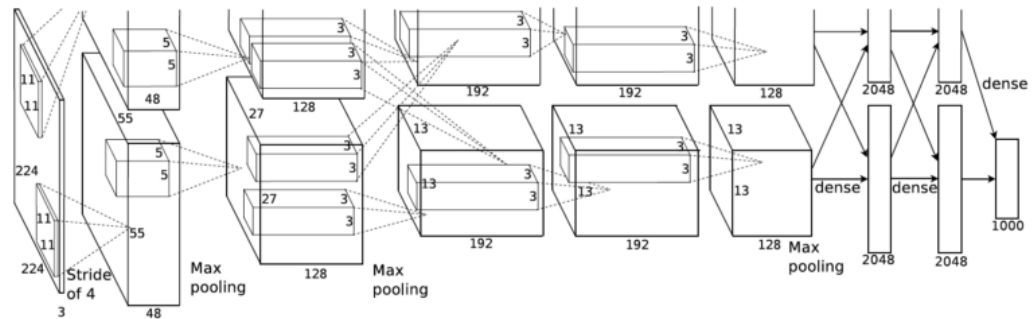


Rysunek 4.2: Architektura sieci LeNet

4.2.2. AlexNet

[Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, 2012] - pierwsza prawdziwa konwolucyjna sieć neuronowa (CNN), która pomogła zmienić opinię na temat uczenia głębokiego. Została zaimplementowana z użyciem biblioteki CUDA. Pierwsza sieć neuronowa, ucząca się z powodzeniem dla dużego zbioru danych. Została wytrenowana na zbiorze danych złożonym z ponad 15 milionów obrazów podzielonych na 22000 klas. W testach top-1 i top-5 uzyskała wartości kolejno 37,5% i 17%, co pozwoliło wygrać konkurs ImageNet Large Scale Visual Recognition Challenge. Sieć neuronowa zawiera 60 milionów parametrów i 650 000 neuronów. Architekturę tworzy 8 warstw, gdzie pierwsze 5 to warstwy konwolucyjne, a pozostałe 3 są warstwami w pełni połączonymi, ta ostatnia ma oczywiście 1000 wyjść z funkcji softmax.

AlexNet znacząco przewyższyła swoją wydajnością poprzednich uczestników i wygrała zawody redukując błąd top-5 do 15,32%. Drugie miejsce to błąd ok 26.2% (nie była to CNN). Sieć jest głęboką modyfikacją architektury Yann'a LeCun'a. AlexNet była zaplanowana na dwie karty graficzne, stąd rozdzielenie przepływu informacji na 2 części. Trenowanie sieci na 2 GPU było nowością na te czasy. Sieć została wytrenowana na zbiorze ImageNet. Do wyliczenia funkcji nieliniowych były używane ReLU (tutaj po raz pierwszy okazało się że ReLU działa dużo szybciej niż tangens hiperboliczny). Sieć o której będą uczyli się studenci Data Science na lekcjach historii. [1]



Rysunek 4.3: Architektura sieci AlexNet

4.2.3. VGG Net

[Simoyan i Zisserman, 2014] - sieć złożona z 16 warstw konwolucyjnych, która charakteryzuje się małymi filtrami i dużą głębokością sieci. W trakcie trenowania sieci wyjście jest ustawione na ustalony rozmiar (224 x 224 x 3). Przetwarzanie wstępne obejmuje odjęcie mediany wartości RGB dla każdego piksela. Zdjęcie jest przetwarzane przez stos warstw konwolucyjnych, gdzie używane są filtry o bardzo małym polu widzenia (3x3) [najmniejszy możliwy rozmiar by móc rozpoznać kierunek]. Operacja Max-pooling jest wykonana na polu 4 pikseli, co pokazuje że jest to pobieranie jak najmniejszych cech z obrazu. Ukryte warstwy są wyposażone w nieliniową funkcję aktywacji ReLU. Po stosie warstw konwolucyjnych, następuje nałożenie 3 warstw w pełni połączonych (to takie duże warstwy zawierające wszystkie cechy?). Pierwsze dwie mają 4096 kanałów, trzecia już tylko 1000 (po jednym kanale na klasę obiektu). Obecnie architektura ta jest dość popularnym wyborem dla wyodrębniania cech ze zdjęć. Konfiguracje wag dla zbioru obrazów z ImageNet są dostępne online. W tej sieci problemem jest

140 milionów parametrów, którymi czasem trzeba zarządzać. [13]

4.2.4. GoogleNet / Inception

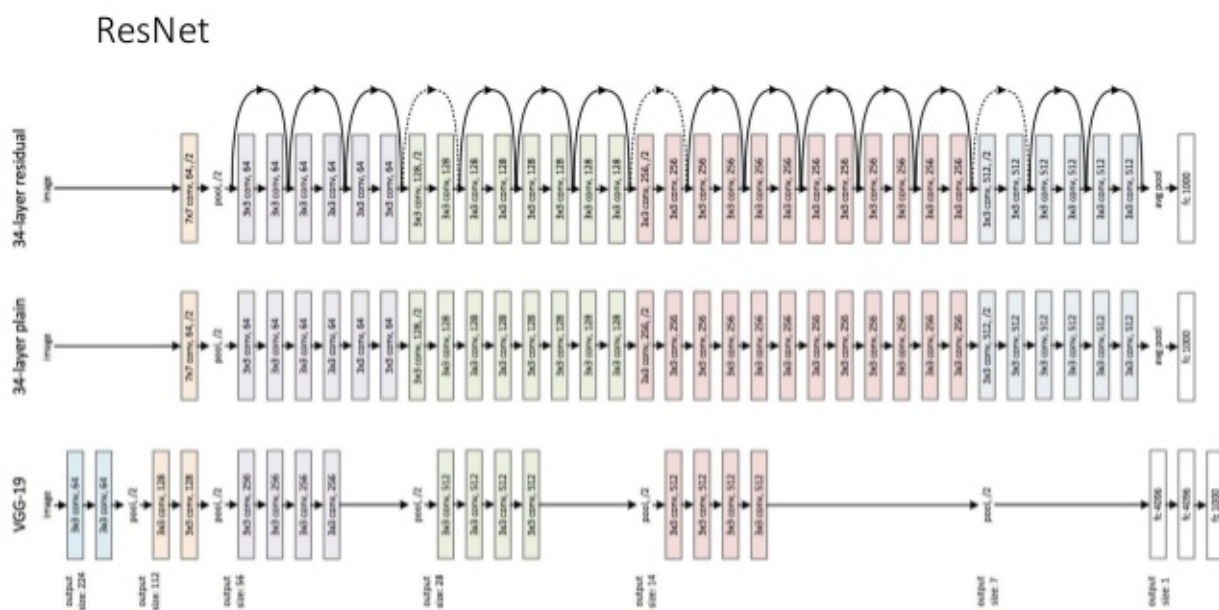
[Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabino-
vich, 2014] – produkt jak nazwa wskazuje wyszedł z firmy Google. Charak-
teryzuje się przełomową poprawą użycia zasobów wewnętrznych. ImageNet
jest zaprojektowana na jak największą szerokość i głębokość jednocześnie
utrzymując stałe zużycie mocy obliczeniowej. Optymalizacja jakości opiera
się na zasadzie Hebbian’a (jest to teoria neuronauki, która mówi o adaptacji
neuronów w trakcie nauki). Błędy klasyfikacji wynoszą top-1: 17,2% i top-
5: 3,58% (dla v3). Sieć składa się łącznie z 22 warstw. Sieć jest modułowa,
gdzie każdy moduł pełni rolę wielopoziomowego ekstraktora cech przelicza-
jącego konwolucje na macierzach rozmiarów 1x1, 3x3, 5x5. Zaletą tej archi-
tektury są bardzo małe rozmiary wag (mniej niż 100MB dla v3). Kolejną
ważną zaletą jest ilość parametrów, tylko 4 miliony co jest wynikiem 12 razy
lepszym od AlexNet. Jest to jedna z najbardziej złożonych sieci względem
architektury modułowej. W wersji naiwnej każda warstwa posiada 4 eks-
traktory cech (3 konwolucyjne i jedną 3x3 max pooling), następnie wartości
te są składane i wysyłane do warstwy wyżej. Do poprawienia wydajności
obliczeniowej pozbyto się warstwy “w pełni połączonej”. [6]

4.2.5. ResNet

[Kaiming He et al, 2015] – architektura oparta na mikro modułach. Charak-
teryzuje się możliwością odrzucania połączeń i posiada ogromny moduł na
normalizację zbioru iteracji (ang. batch normalization). Normalizacja jest
pomysłem zaczerpniętym z rekurencyjnych sieci neuronowych. Ta technika
pozwala stworzyć sieć ze 152 warstwami ukrytymi przy zachowaniu złożo-
ności mniejszej niż VGG. [12]

4.2.6. ResNeXt

[Saining Xie, Ross Girshick, Piotr Dollar Zhuowen Tu Kaiming He, 2017] -
modularyzowalna [DBLP:journals/corr/abs-1805-08700]



Rysunek 4.4: Architektura sieci ResNet

4.2.7. RCNN – obrazek jest zastępczy nie ma nic w internecie interesującego (zrobić własną wizualizację)

[Authors, Year] - [10]

4.2.8. YOLO - You Only Look Once

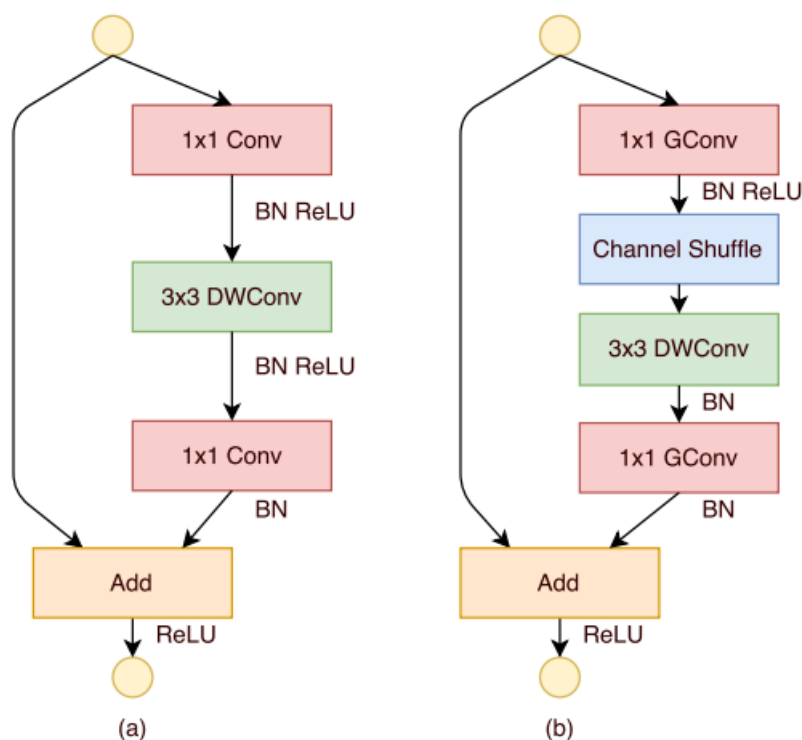
[Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, 2016] - You Only Look Once jest systemem wykrywania obiektów w czasie rzeczywistym. Obraz jest dzielony na części oddzielnymi od siebie prostokątami. Cały przepływ danych do wykrycia obiektów jest jedną siecią, dlatego może być zoptymalizowany [15]

4.2.9. SqueezeNet

[Authors, Year] - [16]

4.2.10. SegNet

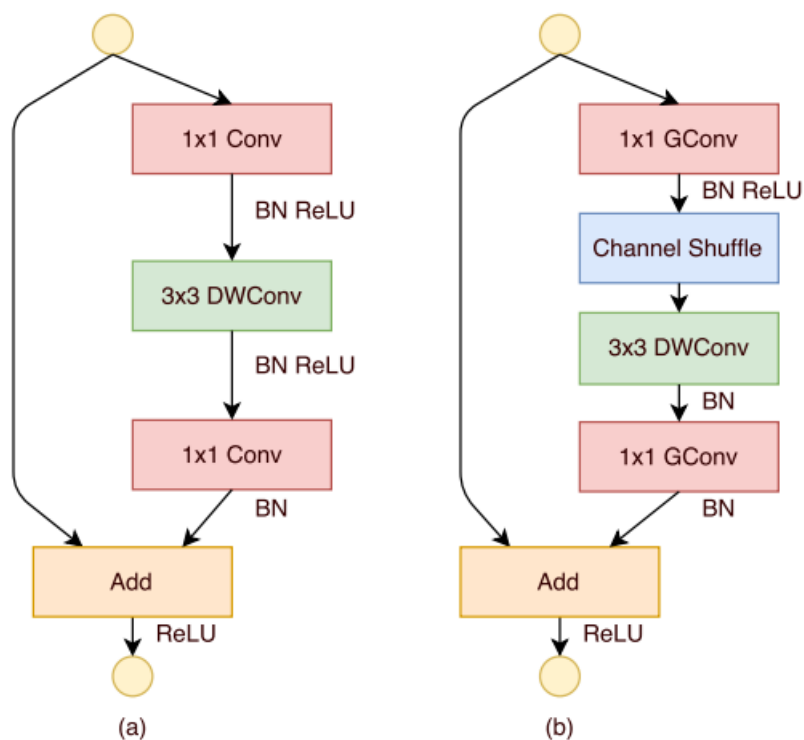
[Authors, Year] - [17]



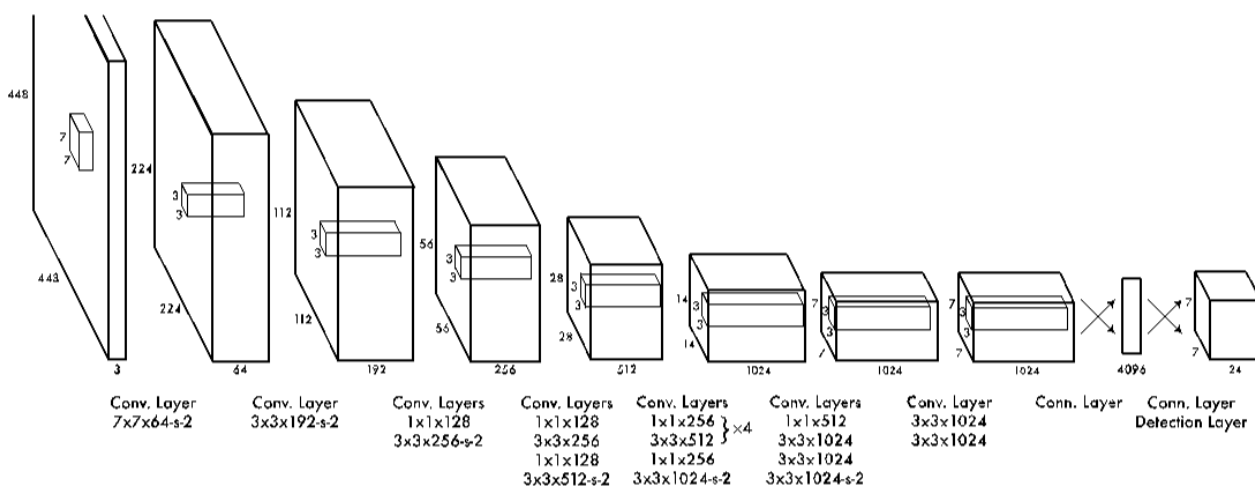
Rysunek 4.5: Architektura sieci ResNeXt

4.2.11. Generative Adversarial Network

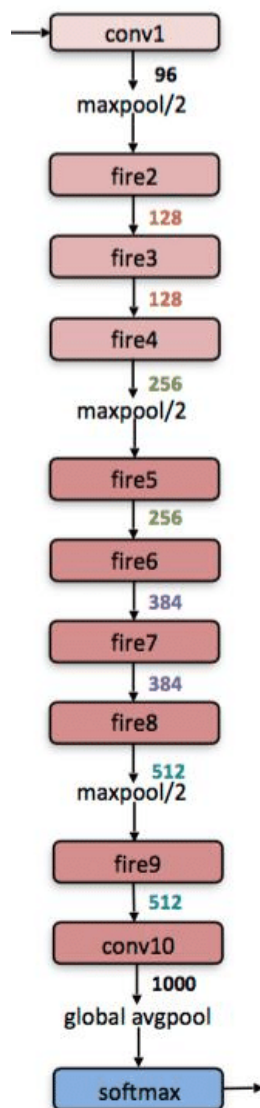
[Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair†, Aaron Courville, Yoshua Bengio‡, 2014] - w skrócie GAN. Są architekturą sieci neuronowych skomponowanych z dwóch sieci przeciwstawionych wobec siebie. Zaprojektowane na uniwersytecie w Montrealu (gdzie powstało większość przełomów dotyczących neuronów) przez największe obecnie autorytety w dziedzinie. GAN obudził duże nadzieje na szybkie i "twórcze" działania algorytmów, jego najczęstszym zastosowaniem jest tworzenie komputerowych "dzieł sztuki". Jak to działa? Jedna sieć generuje kandydatów, a druga ich ocenia wytworzone obiekty. W ten sposób obie sieci uczą się od siebie wzajemnie. - [5]



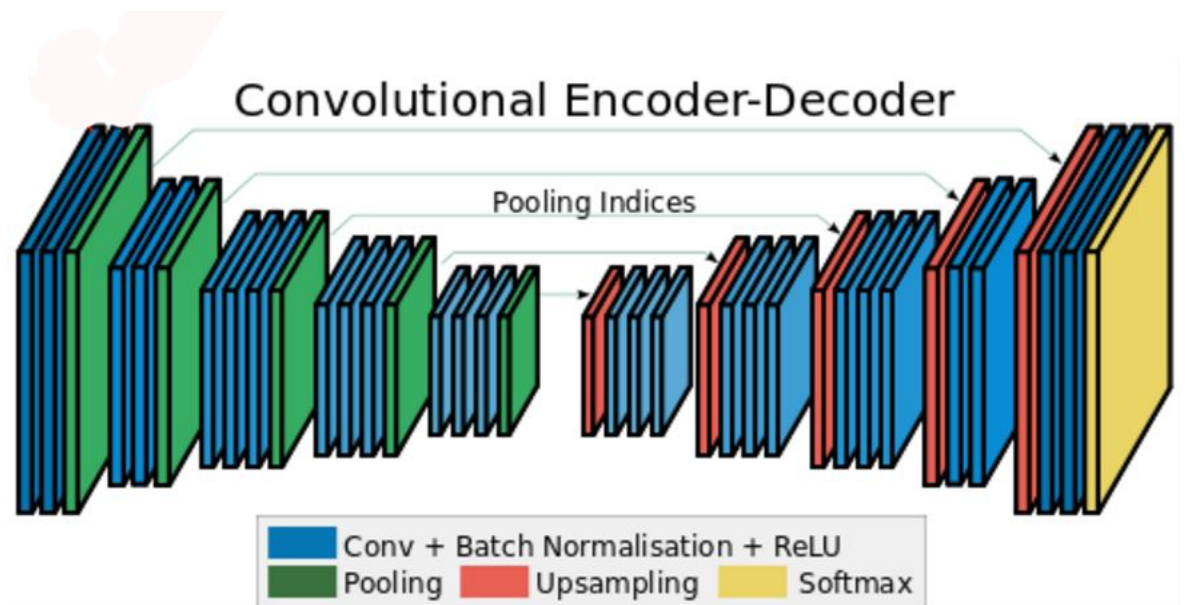
Rysunek 4.6: Architektura sieci RCNN



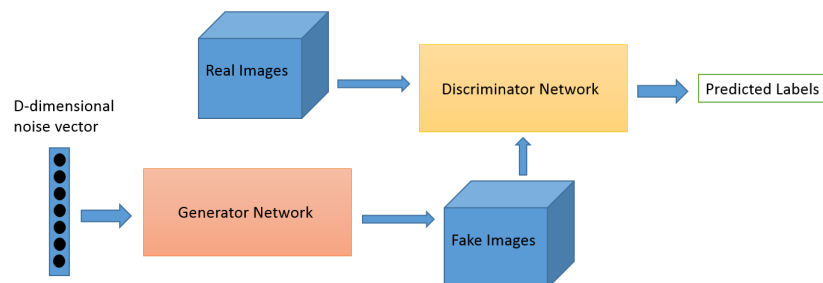
Rysunek 4.7: Architektura sieci YOLO



Rysunek 4.8: Architektura sieci SqueezeNet



Rysunek 4.9: Architektura sieci SegNet



Rysunek 4.10: Architektura sieci GAN

Rozdział 5.

Testy wydajności

Rozdział przedstawia praktyczne wykorzystanie omówionych wcześniej mechanizmów. Autor wybrał kilka z omawianych w poprzednim rozdziale architektur i wykonał testy wydajności na jednym zbiorze danych. Porównanie ma przedstawiać czas uczenia na modelach z wcześniej ustawionymi parametrami.

5.1. Dane

Zdjęcia przedstawiają prześwietlenia płuc przygotowane i przeanalizowane przez zespół złożony z: Daniel S. Kermany, Michael Goldbaum, Wenjia Cai, Carolina C.S. Valentim, Huiying Liang, Sally L. Baxter, Alex McKeown, Ge Yang, Xiaokang Wu, Fangbing Yan, Justin Dong, Made K. Prasadha, Jacqueline Pei, Magdalene Y.L. Ting, Jie Zhu, Christina Li, Sierra Hewett, Jason Dong, Ian Ziyar, Alexander Shi, Runze Zhang, Lianghong Zheng, Rui Hou, William Shi, Xin Fu, Yaou Duan, Viet A.N. Huu, Cindy Wen, Edward D. Zhang, Charlotte L. Zhang, Oulan Li, Xiaobo Wang, Michael A. Singer, Xiaodong Sun, Jie Xu, Ali Tafreshi, M. Anthony Lewis, Huimin Xia, Kang Zhang. Obrazy zostały udostępnione z licencją Creative Commons 4.0. [44]

Dane złożone są z:

- 5862 obrazów,
- podzielone na 2 kategorie,
- rozłożone w 3 katalogach,
 - test – 234 zdrowe płuca, 390 z zapaleniem płuc,
 - train – 1341 zdrowe płuca, 3875 z zapaleniem płuc,

– val – 8 zdrowe płuca, 8 z zapaleniem płuc.

Ilość danych na możliwości głębokiego uczenia sieci neuronowych jest niewielka. Gdyby zwiększyć ilość sygnałów wejściowych kilkunastokrotnie, każda nowoczesna architektura bez problemu uzyskałaby wynik zbliżony do nieomylnego, powyżej 99,5%. Zapalenie wywołane wirusem i bakteriami jest w jednym katalogu, dlatego algorytm powinien nie odróżniać między sobą dwóch stanów chorobowych. Zapalenie płuc może być wywołane również *Mycoplasma* i grzyby, jednak te dwie kategorie nie zostały umieszczone w zbiorze danych. Obrazy rentgenowskie klatki piersiowej (przód-tył) zostały wybrane z kohort pacjentów pediatrycznych w wieku od jednego do pięciu lat z Centrum Medycznego Kobiet i Dzieci w Guangzhou. Wszystkie zdjęcia zostały wykonane w ramach rutynowej opieki klinicznej pacjentów.

Zdjęcia rentgenowskie zostały przygotowane pod kątem analizy. Wszystkie radiogramy klatki piersiowej zostały przeszukane pod kątem kontroli jakości, usuwając wszystkie skany nieczytelne oraz będące niskiej jakości. Diagnozy dotyczące zawartości obrazów zostały ocenione przez zespół dwóch lekarzy specjalistów, a następnie przefiltrowane przez kolejnego lekarza celem usunięcia ewentualnych błędów.

5.2. Testowane architektury

Wybrane architektury na których zestaw został przetestowany:

- ResNet,
- VGG,
- ResNeXt,
- Wide Residual Network,
- Inception,
- DenseNet.

Architektury te zostały wybrane na szereg zastosowań komercyjnych w rozwiązywaniu zawodów na serwisie Kaggle, gdzie uczestnicy wybierają algorytmy z całego repertuaru uczenia maszynowego by zdobyć wysokie nagrody finansowe. Przez ostatnie dwa lata serwis został zdominowany przez algorytmy używające głębokiego uczenia sieci, głównie ze względu

na łatwość implementacji i uniwersalność zastosowań. Osoby, które potrafią świetnie i szybko stroić sieć pod nowe wyzwania, zdobywają wysokie nagrody w zawodach i hackatonach oraz zyskują propozycje zawodowe z wysokim uposażeniem. Kolejną zaletą architektur jest mnogość narzędzi i instrukcji wspierających przenoszenie uczenia na nowe domeny. Popularna sieć neuronowa zawiera często zestaw wskazówek, które warstwy należy doстроить, które wytrenować od nowa, co daje bardzo szybkie przeniesienie z idei do przystępnie działających modeli.

5.2.1. Środowisko obliczeniowe

Sprzęt na którym zostały wykonane prace:

- Intel® Xeon® Processor E5-2630 v4 (10 rdzeni, 20 wątków),
- GeForce GTX 1070 Ti (8Gb).

Narzędzia programistyczne używane do wykonywania obliczeń:

- Linux Mint 18.3 Sylvia (GNU/Linux 4.13.0-41-generic x86_64),
- conda 4.5.11,
- Python 3.6.5,
- Tensorflow 1.9.0,
- Keras 2.2.2,
- PyTorch 0.3.1.post2,
- fastai.

Sposób przygotowania środowiska pracy z danymi w systemie typu UNIX jest identyczny jak dla programistów tego systemu. Należy zainstalować wybraną wersję środowiska Python, dokładna wersja zależy od wybranych bibliotek, jednak obecnie wszystkie rozwijane utrzymują wsparcie w pierwszej kolejności dla wersji Python 3.5 i wyższe. Dla ułatwienia zarządzania paczkami, sugeruje się instalację oprogramowania Anaconda na maszynie roboczej. Jest to zestaw narzędzi i bibliotek używanych do pracy z danymi. Anaconda potrafi tworzyć środowiska wirtualne, zarządzać wersjami bibliotek unikając tzw. piekła zależności i w dodatku nie trzeba mieć uprawnień roota w systemie, co jest częstym problemem podczas korzystania z maszyn

współdzielonych, bez uprawnień administratora nie można zainstalować bibliotek korzystających z urządzeń GPU.

Trudnością w wykonywaniu eksperymentów na sieciach neuronowych jest możliwość replikacji wyników przez innych. Nawet znając założenia eksperymentu i mając ten sam zestaw danych wejściowych dość często można spotkać się z sytuacją kiedy wyniki obliczeń i dokładność będą się trochę różniły między sobą. Taki stan rzeczy wynika ze specyfiki zagadnienia i wpływ mają tutaj wartości losowe wag, wybrane progi, oraz hiperparametry dobierane przy uruchamianiu algorytmu. Wszystkie te elementy mogą doprowadzić do znalezienia zupełnie innego minimum lokalnego na płaszczyźnie badanej przez algorytm. Aby zapobiec rozbieżności wyników między kolejnymi treningami sieci, przedstawia się tutaj szczegółowy opis procesu.

5.3. Proces eksperymentu

Należy zacząć od przygotowania danych, struktury katalogów. Obrazy podzielone są na trzy katalogi, kolejno:

- test,
- train,
- val.

Jest to de facto standard struktury ułożenia danych. W każdym z wyżej wymienionych katalogów, zdjęcia muszą zostać ułożone w odpowiadającej klasie katalogu:

- normal,
- pneumonia.

Podział danych na zestawy treningowe, uczące i walidacyjny zwyczajowo jest w proporcjach 70%/25%/5%. Tutaj podział jest w proporcjach 89%/11,8%/0,2%. Skrypt, który przygotowuje algorytm jest podobny dla wszystkich architektur, jest to naturalne ponieważ określa się w nim hiperparametry dla algorytmu. Po wielokrotnych eksperymentach na różnych parametrach, okazuje się że użyta karta graficzna dość szybko wyczerpuje ograniczenia pamięciowe, a załadowanie całego zbioru danych jest niemożliwe. Zmusza to użytkownika do skalowania zdjęć do dużo niższej rozdzielczości. Najmniejszym wspólnym mianownikiem dla wszystkich zastosowanych

architektur jest przeskalowanie zdjęć do rozmiaru 400 x 400 pikseli pozostawiając 3 warstwy na kolor. Pozostawiona jest również liczba epok dla każdej z architektur, dla wyrównania szans uczenia. Współczynnik uczenia został na sztywno ustawiony na wartość 0,01. W późniejszych badaniach zastosowany został algorytm wygaszania stałej uczenia, czyli zmianę wraz ze znajdowaniem kolejnych minimów lokalnych płaszczyzny funkcji szukanej przez sieć.

Dla uniknięcia tworzenia nowych modeli, co na dostępnym sprzęcie wymagałoby tygodni obliczeń dla każdej z architektur, użyto w eksperymencie modeli uczonych na zawody ImageNet. Każdy model jest uczony na zbiorze 1,2 miliona zdjęć przydzielonych do 1000 klas.

Eksperymenty zostały wykonane w kolejności od najstarszej architektury spełniającej wymóg klasyfikacji obrazów, który został ustalony na powyżej 80%. Następnie każda z architektur przechodzi strojenie indywidualne, tak by uzyskać najwyższy możliwy jej wynik. Zwycięzca otrzymuje możliwość honorowego porównania pomiaru uczenia zarówno na CPU oraz GPU.

5.3.1. Przegląd danych uczących

Na początek warto zaznajomić się z wyglądem sygnałów wejściowych, ich strukturą i parametrami. Czyszczenie danych to zazwyczaj większość procesu pracy z danymi. Tutaj jednak zostały one skatalogowane i przystosowane do eksperymentów. Jednak zawsze należy korzystać z dobrych praktyk i sprawdzić przynajmniej kilka losowych obiektów pod kątem poprawności.

```
~ 'PNEUMONIA ' , 'NORMAL '
~ \[ 'NORMAL2-IM-1440-0001.jpeg ' ,
~ 'NORMAL2-IM-1437-0001.jpeg ' ,
~ 'NORMAL2-IM-1431-0001.jpeg ' ,
~ 'NORMAL2-IM-1436-0001.jpeg ' ,
~ 'NORMAL2-IM-1430-0001.jpeg ' ]
```

Listing 5.1: Podgląd danych

```
PATH = "~/data/chest_xray/"
os.listdir(PATH)
files = os.listdir(f'{PATH}val/normal')[ :5]
```

Listing 5.2: Wyświetlenie struktury danych

Struktura nie budzi podejrzeń. Mając dostęp do maszyny wyposażonej w Jupyter Notebook, można w prosty sposób podejrzeć wybrany obraz. Praktyka ta jest szczególnie zalecana pracując na maszynie lokalnej, jeszcze przez uruchomieniem na środowisku obliczeniowym.

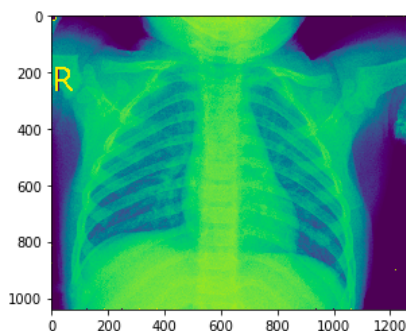
Jak wyglądają dane zdjęcia dla komputera:

```
'img.shape'
(1040, 1272)
'img[:4,:4]'
array([[169, 171, 156, 167],
       [174, 159, 156, 157],
       [168, 158, 167, 255],
       [169, 169, 255,   0]], dtype=uint8)
```

Listing 5.3: Struktura obrazu w formie macierzy

```
plt.imshow (img =
    plt.imread(f'{PATH}val/normal/{files[4]}'))
```

Listing 5.4: Wyświetlenie obrazu w środowisku graficznym



Rysunek 5.1: Losowy plik z zestawu walidacyjnego.

Warto wybrać kilka losowych zdjęć dla potwierdzenia ogólnej poprawności danych. Nie sprawdzi się w ten sposób wszystkich danych, ale pozwoli

to na wstępny ogłęd z czym się pracuje.

5.3.2. Wyniki nauczania

Kolejnym wspólnym elementem dla każdego z programów głębokiego uczenia maszynowego jest przeglądanie danych wynikowych i analiza przypadków w których sieć gorzej sobie radzi albo radzi sobie nadzwyczaj dobrze. Powszechną praktyką jest samodzielne przejrzanie kilku przykładów następujących kategorii:

- Kilka losowo wybranych poprawnie sklasyfikowanych odpowiedzi,
- kilka losowo wybranych niepoprawnie sklasyfikowanych odpowiedzi,
- odpowiedzi dla każdej z klas z najwyższą pewnością (prawdopodobieństwem) poprawnie sklasyfikowane,
- odpowiedzi dla każdej z klas z najwyższą pewnością (prawdopodobieństwem) niepoprawnie sklasyfikowane,
- odpowiedzi najbardziej niepewne, najbliższe przejścia do innej klasy

Wyświetlenie po kilka obrazów z każdej z powyższych kategorii pozwoli zrozumieć dlaczego algorytm zachowuje się w danych sposób i znacząco ułatwia jego modyfikacje. Jak wspomniano we wcześniejszych rozdziałach, uczenie głębokie jest jeszcze praktyką skazaną na metodykę prób i błędów, dlatego wizualizacja jest najprostszym sposobem na uwidocznienie błędów.

5.4. Optymalizacja wyników

5.4.1. Wybór stałej uczenia

Współczynnik uczenia to jeden z najbardziej znaczących hiperparametrów jakie można kontrolować. Definiuje ona tempo aktualizacji parametrów sieci. Jest to jedna z najtrudniejszych wartości do określenia. Do optymalizacji wyników pomocna jest technika opisana przez Leslie N. Smith'a, nazywana cyklicznymi współczynnikami stałej uczenia do trenowania sieci neuronowych (ang. Cyclical Learning Rates for Training Neural Networks). W założeniu technika ta ma całkowicie wyeliminować potrzebę ciągłych eksperymentów w celu znalezienia najlepszych wartości i ustawienia harmonogramu

nastawiania globalnych wartości stałej uczenia. Zamiast stale (monotonicznie) obniżać szybkość uczenia się, metoda ta pozwala cyklicznie modyfikować wartość uczenia pomiędzy ustalonymi wartościami krańcowymi minimum i maksimum. Doświadczenie pokazuje (i testy wykonane w publikacji), że trening wykonany osiąga lepszą dokładność klasyfikacji bez potrzeby ręcznego strojenia i to już w zaledwie kilku iteracjach algorytmu. Technika zmiennych parametrów uczenia potrafi z założenia samodzielnie określić odpowiednie ramy parametrów liniowo zwiększając stopę uczenia przez kilka epok. [42]

5.4.2. Sztuczne zwiększenie ilości danych

Zwiększając ilość epok można dostrzec, że sieć po przekroczeniu pewnej granicy zależnej od ilości posiadanych danych spotyka się z problemem przeuczenia. Oznacza to, że model przestaje uczyć się rozpoznawać konkretne cechy obrazu, a zaczyna wkuwać na pamięć konkretne obrazy ze zbioru treningowego, tracąc umiejętność uogólniania konkretnych klas danych. Doprowadza to do sytuacji kiedy zestaw treningowy jest zapamiętany i rozpoznawany prawidłowo dla każdego obiektu, ale nowe dane, nie spotkane wcześniej przez sieć, są rozpoznawane gorzej niż w przypadku kiedy sieć umiałaby generalizować na podstawie cech obrazu. Dobrym sposobem na uniknięcie problemu przeuczenia jest dostarczenie większej ilości różnorodnych danych. Technika tworzenia nowych danych z istniejących nazywa się powiększaniem danych (ang. data augmentation). Jest to zaawansowany proces, często pomijany w książkach ze względu na trudności w implementacji i zwyczajowy brak gotowych rozwiązań w najpopularniejszych bibliotekach. Proces zwiększania ilości danych w uczeniu maszynowym polega losowej zmianie obrazów w procesie powielania obrazów treningowych. Zmiana jest drobną modyfikacją, nie mającą wpływu na główne cechy w tym ostateczne rozpoznanie obrazu. Najlepszymi operacjami powielającymi dane są:

- poziome i pionowe odbicie (jeśli odwrócony obraz ma sens np. zdjęcia satelitarne),
- zbliżanie,
- obracanie obrazu o kilka stopni (kąt obracania powinien być niewielki).

Konwolucyjna sieć neuronowa jest w stanie nauczyć się lepiej rozpoznawać cechy zdjęcia jeśli otrzyma kilka modyfikowanych kopii tego samego

obrazu. Należy jednak pamiętać że nie jest to metoda pozwalająca uzyskać setki obiektów uczących z każdego przykładu, tutaj znów ryzykuje się przeuczeniem. Zwyczajowo pięć sztuk z każdego obiektu pozwala zwiększyć znacząco zestaw uczący bez ryzyka przeuczenia. Jak widać poniżej na przykładzie architektury ResNet34, dodatnie sztucznie wygenerowanych danych znacząco zwiększyło jakość klasyfikacji, aż o 12,5%. Niestety metoda nie jest jeszcze powszechnie stosowana, wynika to po części braku styczności z niewielką ilością danych badaczy dużych ośrodków takich jak DeepMind czy Microsoft AI Research. W największych laboratoriach programiści mają do dyspozycji ogromne ilości danych i na żądanie otrzymują zasoby na pozyskanie dodatkowych danych z centrów danych. Z tego powodu technika powielania nie ma odzwierciedlenia w bibliotekach rozwijanych przez duże ośrodki badawcze.

5.4.3. Cykliczny współczynnik uczenia

Kolejną techniką poprawienia jakości uczenia jest zastosowanie algorytmu gradientu spadkowego z ponownym uruchamianiem. Metoda ta polega na wygaszaniu współczynnika uczenia, zmniejszając jego wartość w trakcie pracy algorytmu. Następnie wykonywana jest operacja próby wyjścia z minimum lokalnego celem znalezienia prawidłowej przestrzeni wag, takiej która będzie cechowała się stabilnością dla większej gamy przykładów, prócz samej dokładności. Znów celem jest generalizowanie rozpoznawanej klasy, a nie uczenie się wszystkich przykładów. [43]

5.5. Analiza wyników

Najważniejszą miarą wydajności sieci jest wskaźnik jak dobrze klasyfikuje nowe dane. Poważnym błędem jest założenie, że sieć jest w stanie nauczyć się w 100% rozpoznawać wszystkie elementy. Analiza zawiera trzy parametry, które wykazują w jaki sposób konkretne architektury osiągają najwyższe możliwe osiągi na dostępnym sprzęcie. Należy zaznaczyć iż w przypadku kart graficznych z większą ilością pamięci wyniki znacząco uległyby zmianie. Wynika to z faktu iż cały zestaw uczący jest ładowany do pamięci i w przypadku bardziej rozbudowanych sieci, należy obrazy pierw przeskalować tak by nie otrzymać błędu braku dostępnej pamięci. Dzieje się tak szczególnie często, w trakcie stosowania sztucznego powielania danych.

W tej pracy przyjęto dwa scenariusze reakcji na problem braku pa-

mięci. Pierwszy, zmniejszone zostają rozmiary obrazów do rozdzielczości pozwalającej wszystkim architekturom na wytrenowanie od początku do końca z wszystkimi modyfikacjami algorytmu. Metoda najmniejszego wspólnego mianownika pozwala ocenić skuteczność bez uwzględniania ograniczeń sprzętowych. Drugi scenariusz zakłada największą możliwą rozdzielczość tylko dla podstawowego uczenia sieci nie uwzględniając zasobochłonnego strojenia.

Dla pełnej nauki ze strojeniem (pełnym) określone zostały następujące parametry:

- Ilość epok: 6
- Krok uczenia: 0,01
- Rozmiar obrazu: 200 pikseli

Dla podstawowego uczenia sieci bez dodatkowych technik usprawniających model dobrano parametry:

- Ilość epok: 6
- Krok uczenia: zmienny, wyliczany algorytmem dla różnych warstw od 0,001 do 1
- Rozmiar obrazu: 600 pikseli

5.5.1. ResNet

ResNet okazał się architekturą najłatwiejszą do wytrenowania. Jej kompaktowa struktura pozwala zmieścić cały zbiór danych w rozmiarze 500 pikseli również po dodaniu sztucznie wygenerowanych danych. Jest to świetna wiadomość dla osób pracujących na pojedynczych maszynach GPU. Wstępne nauczenie sieci dało świetny rezultat widoczny poniżej.

5.5.2. VGG

5.5.3. ResNeXt

5.5.4. Wide Residual Network

5.5.5. Inception

5.5.6. DenseNet

Najpopularniejsza architektura dla uczenia zdjęć. Jest bardzo szybka do przetrenowania nowych modeli i posiada implementację w każdej bibliotece głębokiego uczenia.

5.5.7. Porównanie czasu uczenia (CPU i GPU)

Sekcja poświęcona na uświadomienie jak wielka przepaść w czasie uczenia istnieje między uczeniem modelu na procesorze jednostki centralnej komputera, a procesorze jednostki graficznej. Trenowanie obrazów odbywa się tylko na ostatnich warstwach architektury ResNet, ponieważ uczenie całości zajęłoby wiele tygodni na zwykłym komputerze klasy PC. Oba pomiary dotyczą już przygotowanej sieci z dostrojeniem, a hiperparametry pozostają takie same, jak w najlepszym wyniku uczenia sieci w poprzednich sekcjach. Do pomiaru czasu użyto prostego narzędzia UNIX'owego "time".

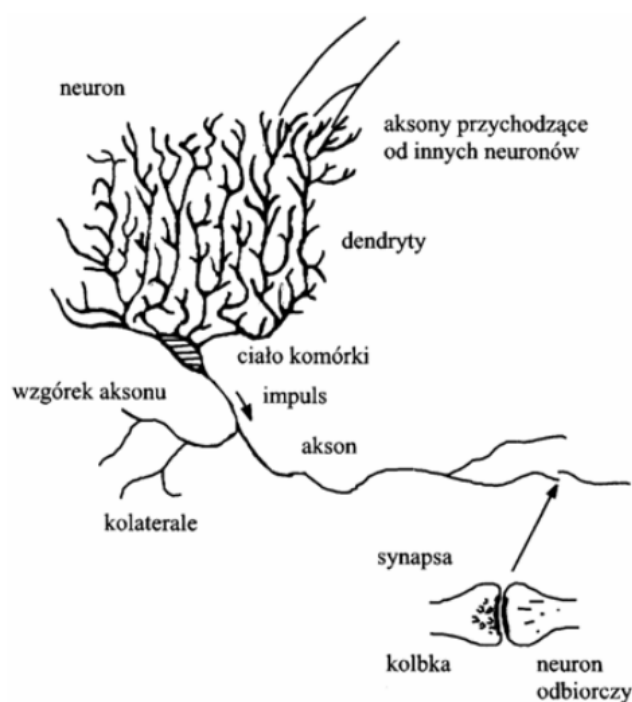
Trening podstawowy: CPU: Epoki: 6, *training_{loss}* : 0.085107*validation_{loss}* : 0.101677, *accuracy* : 1.0*real*0m10.201*suser*2m8.983*ssys*0m5.774*s*

GPU: Epoch: 6, *training_{loss}* : 0.080002*validation_{loss}* : 0.10515, *accuracy* : 0.9375*real*0m12.383*suser*0m12.263*ssys*0m8.115*s*

Cały trening: CPU: Epoch: 6, *training_{loss}* : 0.075047*validation_{loss}* : 0.037492*accuracy* : 1.0*real*43m32.039*suser*496m49.894*ssys*916m41.221*sGPU* :

5.6. Podsumowanie

Pierwszym wnioskiem jaki nasuwa się po przeprowadzeniu testów wydajności jest: nie zaczynać przygody z głębokimi sieciami neuronowymi bez dostępu do procesora graficznego ze średniej półki cenowej. Oczekiwanie na wyniki używając jedynie CPU, zniechęci każdego do eksperymentów, a na tym obecnie polega dobieranie świetnych modeli. Wybierając kartę graficzną również należy się kierować odpowiednio dużą pamięcią. Spora część zbioru



Rysunek 5.2: Wizualizacja uproszczonego modelu neuronu mózgowego.

danych uczących jest ładowana do pamięci GPU, stąd zaleca się minimum 6Gb.

Wpływ wybranej architektury na wynik ma znaczenie, malejące z każdą nową siecią. Zbliżając się do blisko stuprocentowej skuteczności rozpoznawania obrazów, trudno jest stwierdzić że istnieją wyzwania w tej dziedzinie. Należałoby się raczej skupić na bezpieczeństwie sieci i zabezpieczeniami przed próbami oszukania. Ataki stają się popularne, kiedy agenci automatyczni zostają wdrożeni na szeroką skalę.

Dużo ważniejsze od ułożenia kolejnych warstw neuronów okazują się techniki pozwalające optymalizować znajdowanie optymalnego minimum funkcji określającej model. Nowe techniki pojawiające się z każdym miesiącem pozwalają na skuteczne wykorzystanie algorytmu przez osoby nie znające matematyki stojącej za algorytmami głębokiego uczenia.

Podsumowanie

Spis rysunków

1.1.	Wizualizacja uproszczonego modelu neuronu mózgowego. . .	16
1.2.	Wizualizacja prostego sztucznego neuronu.	16
1.3.	Prosta reprezentacja neuronu.	18
1.4.	Prosta sieć neuronowa złożona z dwóch warstw.	19
1.5.	Wykres funkcji sigmoidalnej	23
1.6.	Wykres funkcji tangensu hiperbolicznego	24
1.7.	Wykres rektyfikowanej jednostki liniowej	24
1.8.	Wykres nieszczelnej rektyfikowanej jednostki liniowej	25
1.9.	Wykres obliczeniowy dla funkcji $J(a, b, c) = 3(a + bc)$	27
2.1.	Przykładowa struktura sieci typu feedforward.	31
2.2.	Wartości zerowe na krawędziach macierzy tworzą margines. .	35
2.3.	Operacja łączenia MaxPooling.	36
2.4.	Pojedynczy neuron ze sprzężeniem zwrotnym.	38
2.5.	Wsteczna propagacja błędu w czasie. x - sygnał wejściowy, s - funkcja aktywacji, E - funkcja błędu	40
2.6.	Wizualizacja architektury FCN przeznaczonej do segmentacji obrazu.	41
3.1.	Logo biblioteki Tensorflow	44
3.2.	Logo biblioteki Keras	47
3.3.	Logo biblioteki PyTorch	49
3.4.	Logo biblioteki Tensorflow.js	50
3.5.	Logo biblioteki Paddle	53
3.6.	Logo biblioteki MXNet	54
3.7.	Logo biblioteki Caffe2	56
3.8.	Logo biblioteki ML.NET	58
3.9.	Logo biblioteki CNTK	59
4.1.	Wydajność architektury względem ilości operacji potrzeb- nych do przejścia sygnału przez nią.	64

4.2. Architektura sieci LeNet	65
4.3. Architektura sieci AlexNet	66
4.4. Architektura sieci ResNet	67
4.5. Architektura sieci ResNeXt	68
4.6. Architektura sieci RCNN	69
4.7. Architektura sieci YOLO	70
4.8. Architektura sieci SqueezeNet	71
4.9. Architektura sieci SegNet	72
4.10. Architektura sieci GAN	72
5.1. Losowy plik z zestawu walidacyjnego.	78
5.2. Wizualizacja uproszczonego modelu neuronu mózgowego. . .	84

Bibliografia

- [1] Alex Krizhevsky, Ilya Sutskever i Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. W: *Advances in Neural Information Processing Systems 25*. Wyed. F. Pereira i in. Curran Associates, Inc., 2012, s. 1097–1105. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [2] Ian Goodfellow, Yoshua Bengio i Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [3] Nitish Srivastava i in. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. W: *Journal of Machine Learning Research* 15 (2014), s. 1929–1958. URL: <http://jmlr.org/papers/v15/srivastava14a.html>.
- [4] *Fast.ai Making neural nets uncool again*. URL: <http://www.fast.ai/> (term. wiz. 11.09.2017).
- [5] Ian Goodfellow i in. “Generative Adversarial Nets”. W: *Advances in Neural Information Processing Systems 27*. Wyed. Z. Ghahramani i in. Curran Associates, Inc., 2014, s. 2672–2680. URL: <http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>.
- [6] Christian Szegedy i in. “Going Deeper with Convolutions”. W: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: <http://arxiv.org/abs/1409.4842>.
- [7] Yann Lecun i in. “Gradient-based learning applied to document recognition”. W: *Proceedings of the IEEE*. 1998, s. 2278–2324.
- [8] Adnan Qayyum i in. “Medical Image Retrieval using Deep Convolutional Neural Network”. W: *CoRR* abs/1703.08472 (2017). arXiv: 1703.08472. URL: <http://arxiv.org/abs/1703.08472>.

- [9] Kaiming He i in. “Deep Residual Learning for Image Recognition”. W: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: <http://arxiv.org/abs/1512.03385>.
- [10] Shaoqing Ren i in. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. W: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: <http://arxiv.org/abs/1506.01497>.
- [11] Vinod Nair i Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. W: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, 2010, s. 807–814. URL: <http://dl.acm.org/citation.cfm?id=3104322.3104425>.
- [12] Saining Xie i in. “Aggregated Residual Transformations for Deep Neural Networks”. W: *CoRR* abs/1611.05431 (2016). arXiv: 1611.05431. URL: <http://arxiv.org/abs/1611.05431>.
- [13] Karen Simonyan i Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. W: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: <http://arxiv.org/abs/1409.1556>.
- [14] Matthew D. Zeiler i Rob Fergus. “Visualizing and Understanding Convolutional Networks”. W: *CoRR* abs/1311.2901 (2013). arXiv: 1311.2901. URL: <http://arxiv.org/abs/1311.2901>.
- [15] Joseph Redmon i in. “You Only Look Once: Unified, Real-Time Object Detection”. W: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: <http://arxiv.org/abs/1506.02640>.
- [16] Forrest N. Iandola i in. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size”. W: *CoRR* abs/1602.07360 (2016). arXiv: 1602.07360. URL: <http://arxiv.org/abs/1602.07360>.
- [17] Vijay Badrinarayanan, Ankur Handa i Roberto Cipolla. “SegNet: A Deep Convolutional Encoder-Decoder Architecture for Robust Semantic Pixel-Wise Labelling”. W: *CoRR* abs/1505.07293 (2015). arXiv: 1505.07293. URL: <http://arxiv.org/abs/1505.07293>.
- [18] Ahmed Menshawy Giancarlo Zaccone Md. Rezaul Karim. *Deep Learning with TensorFlow*. Packt Publishing, 2017.

-
- [19] Ryszard Tadeusiewicz. *Odkrywanie właściwości sieci neuronowych przy użyciu programów w języku C#*. 1st. Kraków: Polska Akademia Umiejętności, 2007.
- [20] Shaohuai Shi i in. “Benchmarking State-of-the-Art Deep Learning Software Tools”. W: *CoRR* abs/1608.07249 (2016). arXiv: 1608.07249. URL: <http://arxiv.org/abs/1608.07249>.
- [21] Tianqi Chen i in. “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”. W: *CoRR* abs/1512.01274 (2015). arXiv: 1512.01274. URL: <http://arxiv.org/abs/1512.01274>.
- [22] Yangqing Jia i in. “Caffe: Convolutional Architecture for Fast Feature Embedding”. W: *CoRR* abs/1408.5093 (2014). arXiv: 1408.5093. URL: <http://arxiv.org/abs/1408.5093>.
- [23] Yangqing Jia i in. “Caffe: Convolutional Architecture for Fast Feature Embedding”. W: *arXiv preprint arXiv:1408.5093* (2014).
- [24] Ryszard Tadeusiewicz. *Sieci Neuronowe*. 1st. Warszawa: Akademicka Oficyna Wydawnicza, 1993.
- [25] *The History of Artificial Intelligence*. URL: <https://courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf> (term. wiz. 11.09.2018).
- [26] *Yangqing Jia*. URL: <http://daggerfs.com/> (term. wiz. 11.09.2017).
- [27] *Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification*. URL: <https://data.mendeley.com/datasets/rsrbjbr9sj/2> (term. wiz. 11.09.2018).
- [28] *Caffe2 Deep Learning Framework*. URL: <https://developer.nvidia.com/caffe2> (term. wiz. 11.09.2018).
- [29] *Python vs. R (vs. SAS) – which tool should I learn?* URL: <https://www.analyticsvidhya.com/blog/2017/09/sas-vs-vs-python-tool-learn/> (term. wiz. 20.09.2018).
- [30] *Stop Writing Slow Javascript*. URL: <https://ilikekillnerds.com/2015/02/stop-writing-slow-javascript/> (term. wiz. 11.09.2018).
- [31] *TensorFlow: A system for large-scale machine learning*. URL: <https://www.semanticscholar.org/paper/TensorFlow%3A-A-system-for-large-scale-machine-Abadi-Barham/4954fa180728932959997a4768411ff913> (term. wiz. 11.09.2018).

- [32] *Introducing ML.NET: Cross-platform, Proven and Open Source Machine Learning Framework*. URL: <https://blogs.msdn.microsoft.com/dotnet/2018/05/07/introducing-ml-net-cross-platform-proven-and-open-source-machine-learning-framework/> (term. wiz. 11.05.2018).
- [33] *Deep Learning Libraries by Language*. URL: <http://www.teglor.com/b/deep-learning-libraries-language-cm569/> (term. wiz. 11.09.2019).
- [34] Pal Sujit Gulli Antonio. *Implement neural networks with Keras on Theano and Tensorflow*. 1st. Brimingham, UK: Packt Publishing Ltd., 2017.
- [35] Vishnu Subramanian. *Deep Learning with PyTorch*. Packt, 2018. URL: <http://gen.lib.rus.ec/book/index.php?md5=26f5d5f604442238130e8de8806b01df>.
- [36] *Core Concepts in TensorFlow.js*. URL: <https://js.tensorflow.org/tutorials/core-concepts.html> (term. wiz. 11.09.2018).
- [37] Dr. PKS Prakash; Achyutuni Sri Krishna Rao. *R Deep Learning Cookbook: Solve complex neural net problems with TensorFlow, H2O and MXNet*. Packt Publishing - ebooks Account, 2017. URL: <http://gen.lib.rus.ec/book/index.php?md5=f7e06ac4f57f3f61f78fd488c1133f88>.
- [38] Leif] Leif Larsen [Larsen. *Learning Microsoft Cognitive Services - Second Edition: Leverage Machine Learning APIs to build smart applications*. Packt Publishing, 2017. URL: <http://gen.lib.rus.ec/book/index.php?md5=2ca62cfa2498ca741441b775fe3206b6>.
- [39] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Inc., 2017.
- [40] *Accelerating AI with GPUs: A New Computing Model*. URL: <https://blogs.nvidia.com/blog/2016/01/12/accelerating-ai-artificial-intelligence-gpus/> (term. wiz. 11.09.2018).
- [41] *A BRIEF HISTORY OF NEURAL NETWORK ARCHITECTURES*. URL: <https://www.topbots.com/a-brief-history-of-neural-network-architectures/> (term. wiz. 09.09.2017).
- [42] Leslie N. Smith. "No More Pesky Learning Rate Guessing Games". W: *CoRR* abs/1506.01186 (2015). arXiv: 1506.01186. URL: <http://arxiv.org/abs/1506.01186>.

- [43] Gao Huang i in. “Snapshot Ensembles: Train 1, get M for free”. W: *CoRR* abs/1704.00109 (2017). arXiv: 1704.00109. URL: <http://arxiv.org/abs/1704.00109>.
- [44] *Pneumonia images*. URL: [http://www.cell.com/cell/fulltext/S0092-8674\(18\)30154-5](http://www.cell.com/cell/fulltext/S0092-8674(18)30154-5) (term. wiz. 11.09.2018).
- [45] Sergey Zagoruyko i Nikos Komodakis. “Wide Residual Networks”. W: *CoRR* abs/1605.07146 (2016). arXiv: 1605.07146. URL: <http://arxiv.org/abs/1605.07146>.
- [46] Jonathan Long, Evan Shelhamer i Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation”. W: *CoRR* abs/1411.4038 (2014). arXiv: 1411.4038. URL: <http://arxiv.org/abs/1411.4038>.
- [47] *THE NEURAL NETWORK ZOO*. URL: <https://www.asimovinstitute.org/author/fjodorvanveen/> (term. wiz. 11.12.2017).
- [48] *Gartner Says Deep Learning Will Provide Best-in-Class Performance for Demand, Fraud and Failure Predictions By 2019*. URL: <https://www.gartner.com/en/newsroom/press-releases/2017-09-20-gartner-says-deep-learning-will-provide-best-in-class-performance-for-demand-fraud-and-failure-predictions-by-2019> (term. wiz. 27.10.2017).
- [49] *Recurrent Neural Networks and Long-Short Term Memory*. URL: <https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5> (term. wiz. 11.09.2018).