**Sieć neuronowa dla dużych zbiorów danych**

W poprzedniej części ustanowiliśmy podstawowy fundament naszej podróży w kierunku budowania inteligentnych systemów. Rozróżniliśmy algorytmy uczenia maszynowego w dwóch podstawowych grupach nadzorowanych i nienadzorowanych algorytmów i zbadaliśmy, w jaki sposób model programowania Spark jest przydatnym narzędziem do implementacji tych algorytmów za pomocą prostego interfejsu programowania, wraz z krótkim przeglądem dostępnych bibliotek uczenia maszynowego w Spark. Omówiliśmy również podstawy analizy regresji prostym przykładem i kodem pomocniczym w Spark ML. Tu pokażemy, klaster danych przy użyciu algorytmu K-średnich i głębokiego zanurzenie w królestwie redukcji wymiarowości, co przede wszystkim pomaga nam reprezentować te same informacje o mniejszej liczbie wymiarów bez utraty informacji. Stworzyliśmy podstawę do wdrożenia mechanizmów rekomendacji ze zrozumieniem analizy głównych składników, filtrowania opartego na treści i technik filtrowania grupowego. Po drodze staraliśmy się również zrozumieć niektóre podstawy algebry macierzowej. W tym rozdziale zajmiemy się sieciami neuronowymi i ich ewolucją wraz ze wzrostem mocy obliczeniowej dzięki rozproszonym ramom obliczeniowym. Neuronowe sieci czerpią inspirację z ludzkiego mózgu i pomagają nam rozwiązać niektóre bardzo złożone problemy, które nie są możliwe w przypadku tradycyjnych modeli matematycznych. Tu omówimy:

* Podstawy sieci neuronowych i sztucznych sieci neuronowych

* Modele perceptronowe i liniowe.

* Model nieliniowości

* Sieci neuronowe typu feed-forward.

* Gradientowe zejście, propagacja wsteczna i nadmierne dopasowanie

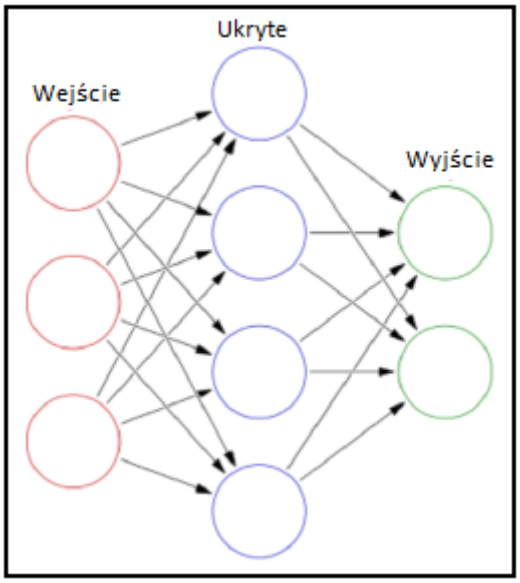
* Nawracające sieci neuronowe

Wyjaśnimy te pojęcia za pomocą łatwych do zrozumienia scenariuszy i odpowiadających im przykładów kodu w Spark ML.

**Podstawy sieci neuronowych i sztucznych sieci neuronowych**

Podstawowe algorytmy i koncepcje modelowania matematycznego omówione w ostatniej części są świetne, jeśli chodzi o rozwiązywanie niektórych ustrukturyzowanych i prostszych problemów. Są prostsze w porównaniu do tego, co ludzki mózg jest w stanie z łatwością zrobić. Na przykład, kiedy dziecko zaczyna identyfikować przedmioty za pomocą różnych zmysłów (wzrok, dźwięk, dotyk itd.), uczy się o tych obiektach w oparciu o niektóre podstawowe elementy składowe ludzkiego mózgu. Istnieje podobny mechanizm we wszystkich żywych istotach z różnicą w poziomie wyrafinowani oparte na cyklu ewolucji. Badanie neurologiczne mózgów różnych zwierząt i ludzi ujawnia, że podstawowymi elementami składowymi mózgu są neurony. Te neurony biologiczne są ze sobą połączone i są zdolne do jednoczesnego przesyłania sygnałów do tysięcy połączonych neuronów. Zauważono, że u bardziej złożonych gatunków, takich jak ludzie, mózg zawiera więcej neuronów niż mniej złożone gatunki. Na przykład uważa się, że ludzki mózg zawiera 100 miliardów połączonych neuronów. Naukowcy odkryli bezpośrednią korelację między ilością i poziomem wzajemnych połączeń między neuronami a inteligencją u różnych gatunków. Doprowadziło to do opracowania sztucznych sieci neuronowych (ANN), które mogą rozwiązać bardziej złożone problemy, takie jak rozpoznawanie obrazów. ANN oferują alternatywne podejście do obliczeń i zrozumienia ludzkiego mózgu. Chociaż nasze rozumienie dokładnego funkcjonowania ludzkiego mózgu jest ograniczone, zastosowanie ANN do rozwiązywania

złożonych problemów przyniosło jak dotąd zachęcające wyniki przede wszystkim w opracowaniu maszyny, która uczy się samodzielnie na podstawie danych kontekstowych, w przeciwieństwie do tradycyjnych obliczeń i algorytmiki podejście. W naszym dążeniu do rozwijania inteligencji poznawczej dla maszyn musimy pamiętać, że sieci neuronowe i obliczenia algorytmiczne nie konkurują ze sobą, lecz się uzupełniają. Istnieją zadania bardziej dostosowane do podejścia algorytmicznego niż sieć neuronowa. Musimy ostrożnie wykorzystać oba te rozwiązania, aby rozwiązać określone problemy. Istnieje wiele systemów, w których wymagamy kombinacji obu podejść. Podobnie jak neurony biologiczne, ANN mają jednostki wejściowe i wyjściowe. Prosty ANN jest reprezentowany w następujący sposób:



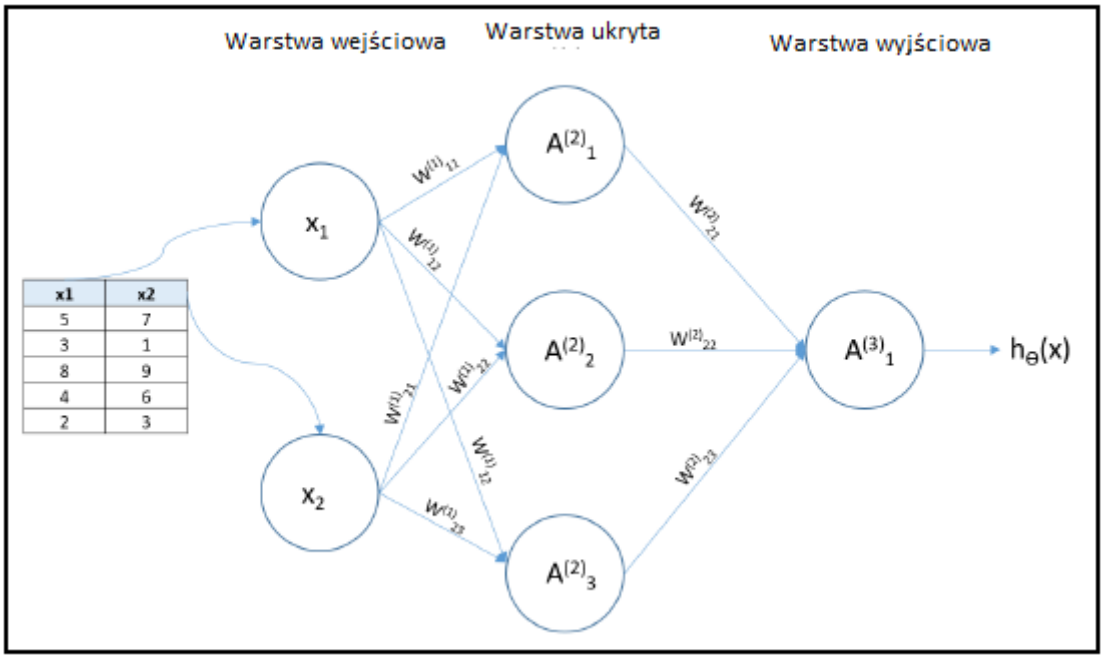

ANN składa się z jednej warstwy wejściowej, która dostarcza dane wejściowe do sieci, jednej warstwy wyjściowej, która reprezentuje ukończone obliczenia ANN oraz jednej lub więcej (zależnie od złożoności) ukrytych warstw, czyli tam, gdzie ma miejsce faktyczne obliczenie i implementacja logiki . Teoria sieci neuronowych nie jest nowa. Jednak w momencie powstania zasoby obliczeniowe, a także zestawy danych były ograniczone, aby wykorzystać pełny potencjał ANN. Jednak wraz z pojawieniem się technologii dużych zbiorów danych i masowo równoległych struktur przetwarzania rozproszonego jesteśmy w stanie zbadać moc ANN dla niektórych innowacyjnych przypadków użycia i rozwiązać niektóre z najtrudniejszych problemów, takich jak rozpoznawanie obrazów i przetwarzanie języka naturalnego. W następnych sekcjach zajmiemy się w głębi ANN za pomocą kilku prostych do zrozumienia przykładów.

**Modele perceptronowe i liniowe**

Rozważmy przykład problemu regresji, w którym mamy dwie zmienne wejściowe i jedną zmienną wyjściową lub zależną, i zilustrujmy użycie ANN do stworzenia modelu, który może przewidzieć wartość zmiennej wyjściowej dla zestawu zmiennych wejściowych:

| x1 | x2 | y |
|---|---|---|
| 5 | 7 | 10 |
| 3 | 1 | 7 |
| 8 | 9 | 12 |
| 4 | 6 | 9 |
| 2 | 3 | 5 |
| 6 | 10 | ? |

W tym przykładzie mamy x1 i x2 jako zmienne wejściowe, a y jako zmienną wyjściową. Dane treningowe składają się z pięciu punktów danych i odpowiednich wartości zmiennej zależnej, y. Celem jest przewidzenie wartości y, gdy x1 = 6 i x2 = 10. Każda dana funkcja ciągła może być zaimplementowana dokładnie przez trójwarstwową sieć neuronową z n neuronów w warstwie wejściowej, 2n + 1 neuronów w warstwie ukrytej i m neuronów w ukrytej warstwie. Przedstawmy to za pomocą prostej sieci neuronowej:




**Notacje składowych sieci neuronowej**

Istnieje znormalizowany sposób oznaczania sieci neuronowych w następujący sposób:

* $x_1$ i $x_2$ są wejściami (możliwe jest również wywołanie funkcji aktywacji na warstwie wejściowej)

* W tej sieci są trzy warstwy: warstwa wejściowa, warstwa wyjściowa i warstwa ukryta.

* W warstwie wejściowej znajdują się dwa neurony odpowiadające zmiennym wejściowym. Pamiętaj, że do ilustracji służą dwa neurony. Jednak w rzeczywistości będziemy mieli setki tysięcy wymiarów, a tym samym zmiennych wejściowych. Podstawowe pojęcia ANN mają teoretycznie zastosowanie do dowolnej liczby zmiennych wejściowych.

* W ukrytej warstwie (warstwa 2) znajdują się trzy neurony: ($a^2_1$, $a^2_2$, $a^2_3$).

*Neuron w końcowej warstwie wytwarza wynik $A^3_1$.

*$a^{(}_{j)i}$: reprezentuje aktywację (wartość obliczaną i wyprowadzaną przez węzeł) jednostki i w warstwie j. Funkcja aktywacji węzła określa dane wyjściowe węzła dla zestawu danych wejściowych. Najprostszą i najczęstszą funkcją aktywacji jest funkcja binarna reprezentująca dwa stany wyjścia neuronu, niezależnie od tego, czy neuron jest aktywowany (odpalanie), czy nie:

Na przykład, $a^2_1$ to aktywacja pierwszej jednostki w drugiej warstwie

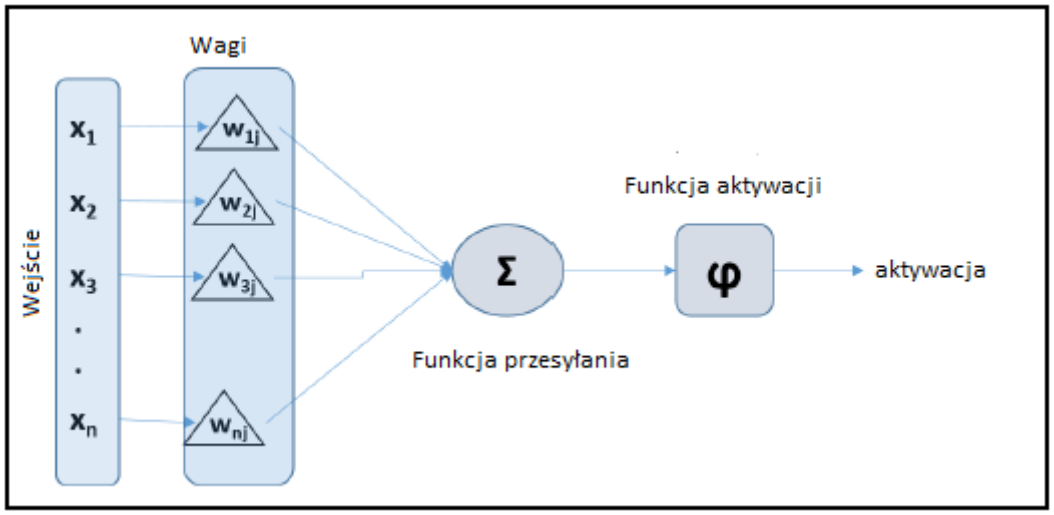
$W^{(l)}_{ij}$ oznacza ciężar złącza, l jest warstwą, z której przesuwa się sygnał, i reprezentuje liczbę neuronów, z której się przemieszczamy, a j reprezentuje liczbę neuronów w następnej warstwie, do której sygnał

przeprowadza się. Wagi są używane do zmniejszania różnicy między rzeczywistą a pożądaną wydajnością ANN:

*Na przykład $W^{(1)}_{12}$ reprezentuje wagę połączenia między dwoma neuronami od warstwy 1 do warstwy 2 dla pierwszego neuronu w warstwie 1 i w kierunku drugi neuron w warstwie 2.

**Matematyczne przedstawienie prostego modelu perceptronu**

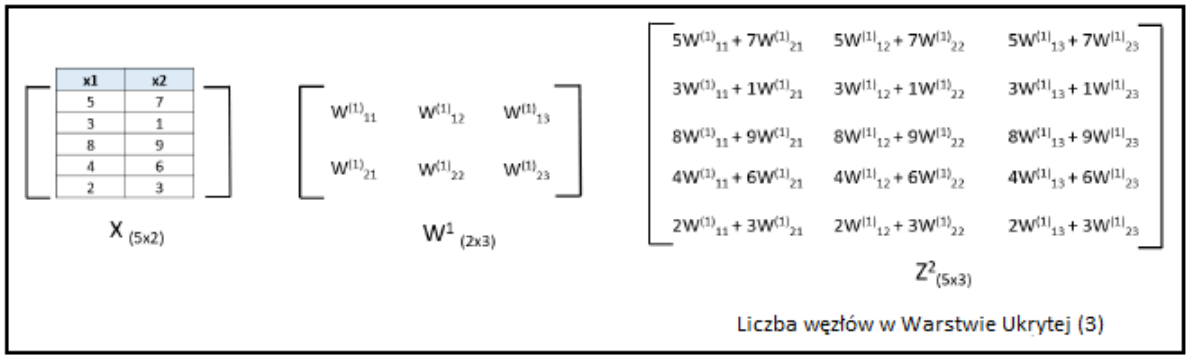
Wydajność sieci neuronowej zależy od wartości wejściowych, funkcji aktywacyjnych na każdym z neuronów i ciężaru na połączeniach. Celem jest znalezienie odpowiednich wag dla każdego połączenia, aby dokładnie przewidzieć wartość wyjściową. Korelację między danymi wejściowymi, wagami, funkcjami przesyłania i aktywacji można wizualizować w następujący sposób:




Podsumowując, w ramach ANN dokonujemy sumy iloczynów danych wejściowych (X) do ich wag (W) i stosujemy funkcję aktywacji f(x), aby uzyskać wynik warstwy przekazywanej jako dane wejściowe do innej warstwy. Jeśli nie ma funkcji aktywacji, korelacja między wartościami wejściowymi i wyjściowymi będzie funkcją liniową. Perceptron jest najprostszą formą ANN używaną do klasyfikacji zestawów danych, które można rozdzielić liniowo. Składa się z pojedynczego neuronu o różnych masach i jednostkach odchylenia. Prosty model perceptronu możemy przedstawić jako iloczyn kropkowy:

$$\varnothing(\sum_{i}^{n} x.w)$$

Ponieważ w naszym przykładzie mamy wiele wartości $x_1$ i $x_2$, obliczenia najlepiej wykonać mnożąc macierz, aby wszystkie funkcje przesyłania i aktywacji mogły zostać obliczone równolegle. Interfejsy API modelu matematycznego są znacznie dostrojone, aby wykorzystywać moc rozproszonych równoległych struktur obliczeniowych w celu wykonywania mnożenia macierzy. Rozważmy teraz nasz przykład i przedstawmy go za pomocą notacji macierzowych. Zestaw danych wejściowych można przedstawić jako x. W naszym przykładzie jest to matryca (5x2). Odważniki można przedstawić jako $W^1_{2x3)}$. Powstała macierz ($Z_2$) jest matrycą 5 na 3, która jest aktywnością drugiej (ukrytej) warstwy. Każdy wiersz odpowiada zestawowi wartości wejściowych, a każda kolumna reprezentuje funkcję przenoszenia lub aktywność na każdym z węzłów w ukrytej warstwie. Można to zilustrować na poniższym schemacie:



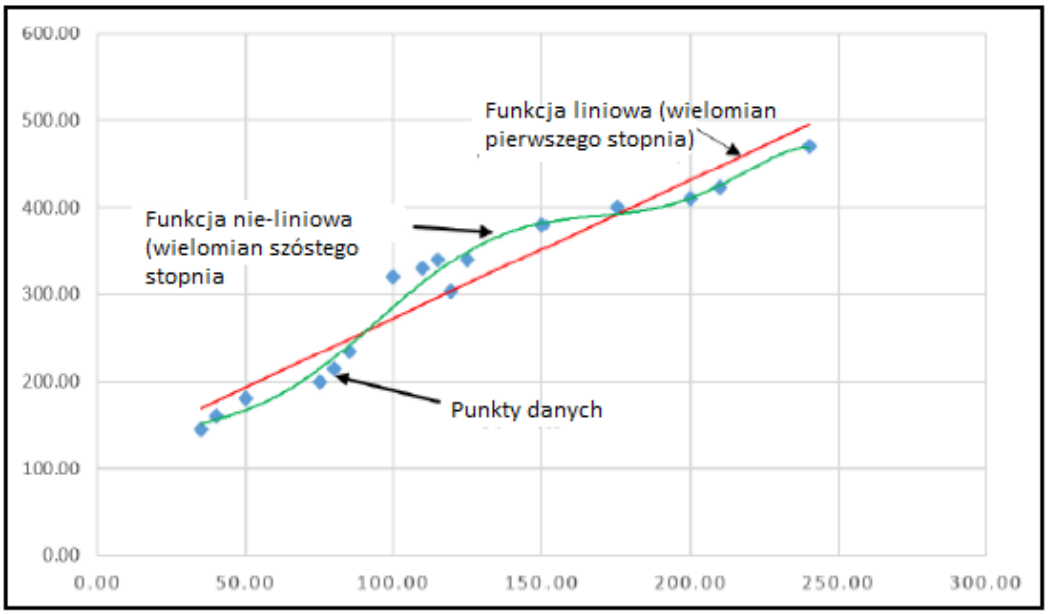

Dzięki temu mamy naszą pierwszą formułę dla sieci neuronowej. Notacja macierzowa jest w tym przypadku bardzo przydatna, ponieważ pozwala nam wykonywać złożone obliczenia w jednym kroku:

$Z^{(2)} = XW^{(1)}$

Dzięki tej formule sumujemy iloczyn danych wejściowych i odpowiadające im wagi synaps dla każdego zestawu danych wejściowych. Dane wyjściowe warstwy są uzyskiwane przez zastosowanie funkcji aktywacji dla wszystkich indywidualnych wartości dla węzła. Głównym celem funkcji aktywacyjnej jest konwersja sygnału wejściowego węzła na sygnał wyjściowy. Podobnie jak w neuronie biologicznym, wynik po zastosowaniu funkcji aktywacji wskazuje, czy neuron jest odpalany, czy nie. Zanim przejdziemy do kolejnych kroków w naszym liniowym modelu perceptronowym, szybko zrozumiemy niektóre z najpopularniejszych funkcji aktywacyjnych używanych w sieciach neuronowych.

**Funkcje aktywacji**

Bez funkcji aktywacji wyjście będzie funkcją liniową wartości wejściowych. Funkcja liniowa to równanie liniowe lub równanie wielomianowe pierwszego stopnia. Równanie liniowe reprezentuje najprostszą formę modelu matematycznego i nie jest reprezentatywne dla rzeczywistych scenariuszy. Nie może odwzorować korelacji w obrębie kompleksu zestawu danych. Bez funkcji aktywacji sieć neuronowa będzie miała bardzo ograniczone możliwości uczenia się i modelowania nieustrukturyzowanych zestawów danych, takich jak obrazy i filmy. Różnicę między funkcją liniową a nieliniową ilustruje poniższy schemat:



Jak widzimy, model liniowy, który otrzymujemy bez użycia funkcji aktywacji, nie może dokładnie modelować danych treningowych, podczas gdy wielomianowe równanie wielomianowe może dokładnie modelować dane treningowe. Za pomocą funkcji aktywacji nieliniowej możemy wygenerować mapowanie nieliniowe między zmiennymi wejściowymi i wyjściowymi oraz modelować złożone scenariusze rzeczywiste. Istnieją trzy podstawowe funkcje aktywacyjne stosowane w każdym neuronie w sieci neuronowej:

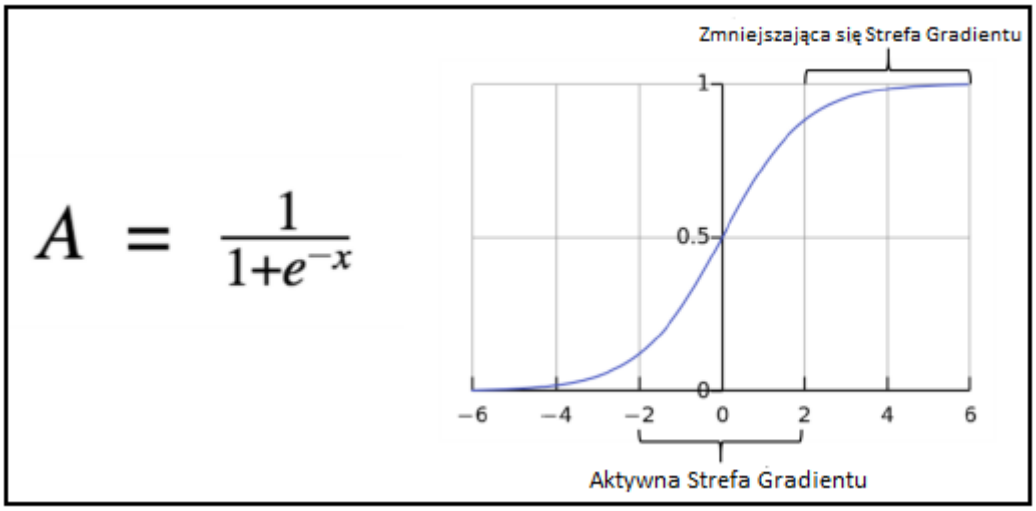
* Funkcja Sigmoid

* Funkcja Tanh

* Zrektyfikowana jednostka liniowa

**Funkcja sigmoidalna**

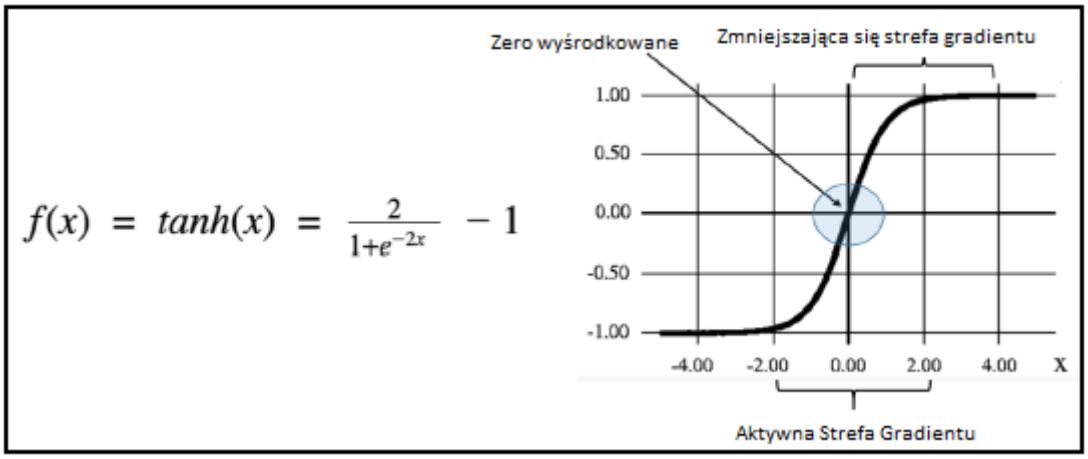
Funkcja sigmoidalna jest jedną z najpopularniejszych funkcji nieliniowych; wyprowadza 0 lub 1 dla dowolnej x wartości wejściowej od -∞ do + ∞. Funkcję można wyrazić matematycznie i graficznie w następujący sposób:



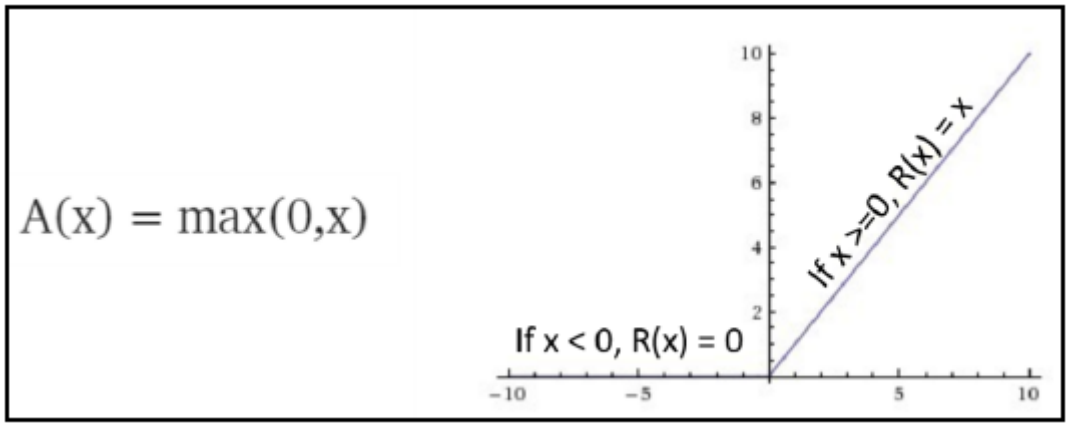

Krzywa funkcji przyjmuje kształt litery S i stąd nazwa sigmoid. Jak widać w tym przykładzie, dla wartości x między -2 a +2 wartości wyjściowe Y są bardzo wysokie. Niewielka zmiana wartości X w tym regionie znacząco przyczynia się do wartości produkcji. Można to nazwać aktywną strefą gradientu. Dla uproszczenia załóżmy ,że rozumiem to jako region na krzywej o najwyższym nachyleniu. Ponieważ wartości X zwykle mieszczą się w zakresie od -∞ do + ∞, krzywa wchodzi w strefę malejącego gradientu. W tym regionie znacząca zmiana wartości X nie ma proporcjonalnego wpływu na wartość wyjściową. Powoduje to zniknięcie problemu z gradientem, gdy model próbuje się zbiegać. W tym momencie sieć nie uczy się dalej lub staje się bardzo wolna i nie ma możliwości obliczeń, aby się zbiegać. Najlepszą częścią funkcji aktywacji sigmoidalnej jest to, że zawsze wyprowadza wartość 0 lub 1, niezależnie od wartości wejściowej X. To sprawia, że jest to idealny wybór jako funkcja aktywacji dla problemów z klasyfikacją binarną. Na przykład doskonale nadaje się do identyfikowania transakcji jako oszukańczej lub nieuczciwej. Innym problemem związanym z funkcją sigmoidalną jest to, że nie jest ona wyśrodkowana na zero (0 <Wyjście < 1). Optymalizacja obliczeń sieci neuronowej jest trudna. Wadę tę eliminuje funkcja tanh.

**Funkcja Tanh**

Funkcja stycznej hiperbolicznej (tanh) jest niewielką odmianą funkcji sigmoidy, która jest wyśrodkowana na 0. Funkcję można przedstawić matematycznie i graficznie w następujący sposób:

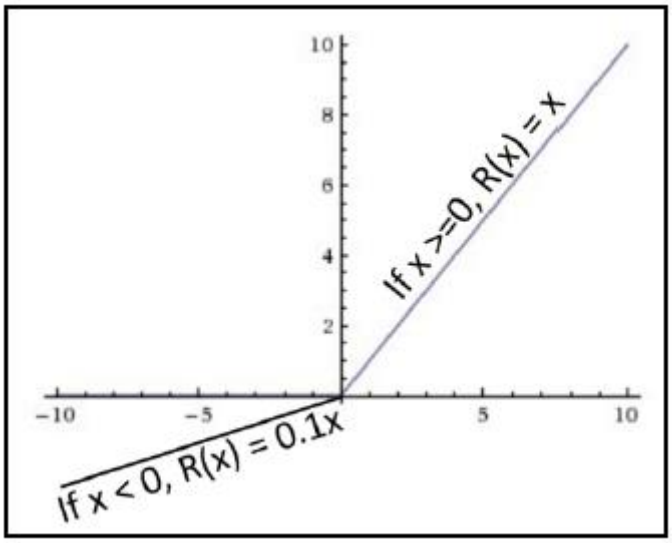



Zakres funkcji tanh wynosi od -1 do 1 i jest wyśrodkowany na zerze; -1 < Wyjście <1. W tym przypadku optymalizacja jest łatwa, a ta funkcja aktywacji jest lepsza niż funkcja sigmoidalna. Jednak funkcja tanh również zanika z powodu zanikającego gradientu ,problem podobny do funkcji sigmoidalnej. Aby pokonać to ograniczenie, używana jest funkcja aktywacji rektyfikowanych jednostek liniowych ReLu. ReLu Funkcja ReLu jest przedstawiona matematycznie i graficznie w następujący sposób:




Forma matematyczna tej funkcji aktywacji jest bardzo prosta w porównaniu z funkcjami sigmoidalnymi lub tanh i wygląda jak funkcja liniowa. Jest to jednak funkcja nieliniowa, która jest obliczeniowa prosta i wydajna, dlatego jest wdrażana w głębokich sieciach neuronowych (sieciach neuronowych z wieloma ukrytymi warstwami). Ta funkcja aktywacji eliminuje problem znikającego gradientu. Ograniczeniem używania ReLu jest to, że możemy go używać tylko do ukrytych warstw. Warstwa wyjściowa musi używać różnych funkcji w przypadku problemów z regresją i klasyfikacją. Funkcja ReLu upraszcza i optymalizuje sieć neuronową obliczenia i konwergencja w porównaniu do funkcji sigmoidalnych i tanh. W przypadku funkcji sigmoidalnej i tanh wszystkie neurony w ukrytych jednostkach strzelają podczas zbieżności modelu. Jednak w przypadku ReLu niektóre neurony będą nieaktywne (dla ujemnych wartości wejściowych), a zatem aktywacje są rzadkie i skuteczne. Chociaż pożądana jest wydajność ze względu na poziomą linię aktywacji, wprowadza ona problem umierania ReLu. Neurony, które przechodzą w stan z powodu ujemnych wartości x, nie reagują na zmiany wartości błędu lub wartości wejściowych, które powodują, że większa część sieci neuronowej jest pasywna. Ten niepożądany efekt uboczny ReLu jest eliminowany przez niewielką zmianę ReLu, zwaną nieszczelną ReLu. W przypadku

nieszczelnego ReLu linia pozioma jest przekształcana w lekko nachyloną linię niehorizontal (0,001 x dla x < 0), zapewniając, że aktualizacje wartości wejściowych po ujemnej stronie widma są aktywne. Nieszczelny ReLu jest przedstawiony graficznie w następujący sposób:




**Model nieliniowości**

Dzięki podstawowym informacjom o funkcjach aktywacyjnych rozumiemy teraz, dlaczego potrzebujemy nieliniowości w sieci neuronowej. Nieliniowość jest niezbędna do modelowania złożonych wzorców danych, które dokładnie rozwiązują problemy regresji i klasyfikacji. Wróćmy jeszcze raz do naszego początkowego problemu przykładowego, w którym ustaliliśmy aktywność ukrytej warstwy. Zastosujmy funkcję aktywacji sigmoidalnej do działania dla każdego z węzłów w ukrytej warstwie. To daje naszą drugą formułę w modelu perceptronu:

$*Z^{(2)} = XW^{(1)}$

$*a^{(2)} = f(z^{(2)})$

Po zastosowaniu funkcji aktywacji f wynikowa macierz będzie miała taki sam rozmiar jak $z^{(2)}$. Oznacza to, że 5 x 3. Kolejnym krokiem jest pomnożenie aktywności warstwy ukrytej przez wagi na synapsie na warstwie wyjściowej. Zobacz schemat notacji ANN. Zauważ, że mamy trzy wagi, po jednej dla każdego łącza z węzłów w ukrytej warstwie do warstwy wyjściowej. Nazwijmy te wagi $W^{(2)}$. Dzięki temu aktywność warstwy wyjściowej można wyrazić za pomocą naszej trzeciej funkcji:

$*Z^{(3)} = a^{(2)} W^{(2)}$

Jak wiemy, $a_{(2)}$ to macierz 5 x 3, a $W^{(2)}$ to macierz 3 x 1. Zatem $Z^{(3)}$ będzie macierzą 5 x 1. Każdy wiersz reprezentujący wartość aktywności odpowiada każdemu indywidualnemu wpisowi w zbiorze danych szkoleniowych.

Na koniec, stosujemy funkcję aktywacji sigmoidalnej do $Z^{(3)}$ w celu uzyskania oszacowania wartości wyjściowej na podstawie zestawu danych treningowych:

$y^{\wedge} = f(Z^{(3)})$

Zastosowanie funkcji aktywacji w warstwie ukrytej i wyjściowej zapewnia nieliniowość w modelu, a my możemy modelować nieliniowy zbiór danych szkoleniowych w ANN.

**Sieci neuronowe ze sprzężeniem zwrotnym**

ANN, o którym mówiliśmy do tej pory, nazywa się neuronową siecią zwrotną, ponieważ połączenia między jednostkami i warstwami nie tworzą cyklu i poruszają się tylko w jednym kierunku (od warstwy wejściowej do warstwy wyjściowej).

Zaimplementujmy przykład sieci neuronowej z prostym kodem Spark ML:

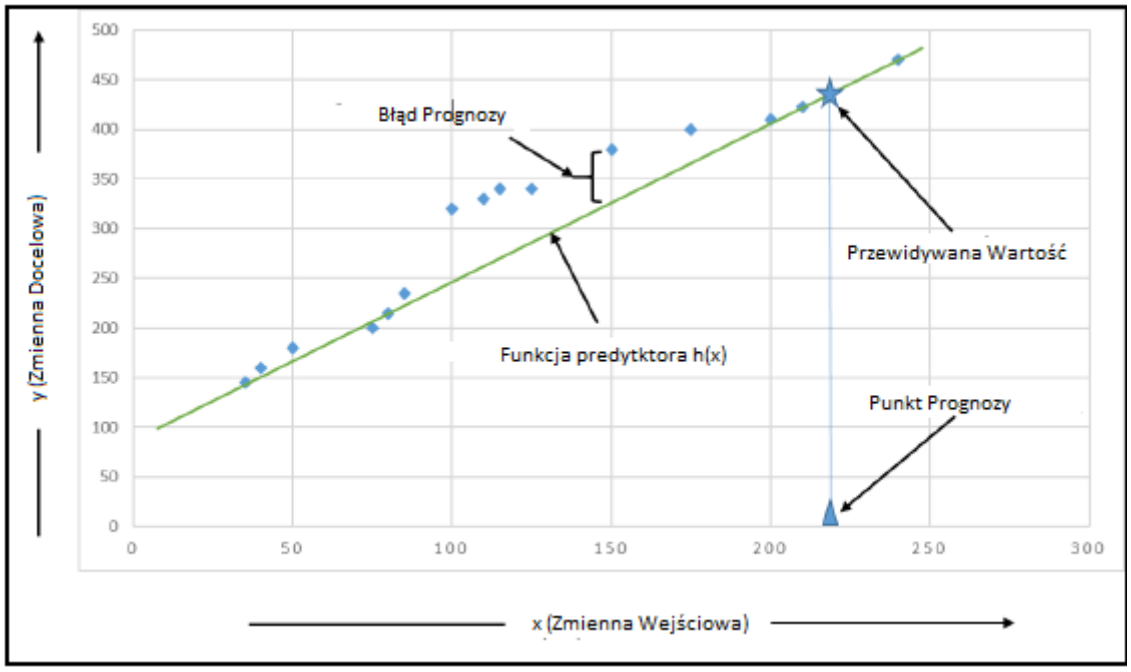
```
object FeedForwardNetworkWithSpark {

def main(args:Array[String]): Unit ={

val recordReader:RecordReader = new CSVRecordReader(0,",")

val conf = new SparkConf()

.setMaster("spark://master:7077")

.setAppName("FeedForwardNetwork-Iris")

val sc = new SparkContext(conf)

val numInputs:Int = 4

val outputNum = 3

val iterations =1

val multiLayerConfig:MultiLayerConfiguration = new

NeuralNetConfiguration.Builder()

.seed(12345)

.iterations(iterations)

.optimizationAlgo(OptimizationAlgorithm

.STOCHASTIC_GRADIENT_DESCENT)

.learningRate(1e-1)

.l1(0.01).regularization(true).l2(1e-3)

.list(3)

.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(3)

.activation("tanh")

.weightInit(WeightInit.XAVIER)

.build())

.layer(1, new DenseLayer.Builder().nIn(3).nOut(2)

.activation("tanh")

.weightInit(WeightInit.XAVIER)

.build())
```

```
.layer(2, new
OutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
.weightInit(WeightInit.XAVIER)
.activation("softmax")
.nIn(2).nOut(outputNum).build())
.backprop(true).pretrain(false)
.build
val network:MultiLayerNetwork = new
MultiLayerNetwork(multiLayerConfig)
network.init
network.setUpdater(null)
val sparkNetwork:SparkDl4jMultiLayer = new
SparkDl4jMultiLayer(sc,network)
val nEpochs:Int = 6
val listBuffer = new ListBuffer[Array[Float]]()
(0 until nEpochs).foreach{i =>
val net:MultiLayerNetwork =
sparkNetwork.fit("file:///<path>/
iris_shuffled_normalized_csv.txt",4,recordReader)
listBuffer +=(net.params.data.asFloat().clone())
}
println("Parameters vs. iteration Output: ")
(0 until listBuffer.size).foreach{i =>
println(i+"\t"+listBuffer(i).mkString)}
}
}
```

Jak widzimy, wartość wyjściowa przewidywana przez nasz model nie jest dokładna. Wynika to z tego, że inicjalizujemy wagi losowo i tylko raz propagujemy do przodu. Potrzebujemy naszej sieci neuronowej, aby zoptymalizować wagi każdego z połączeń między warstwą wejściową a warstwą ukrytą do końcowej warstwy wyjściowej. Osiąga się to dzięki technice zwanej propagacją wsteczną, którą omówimy później.

**Spadek gradientu i propagacja wsteczna**

Rozważmy następujący przykład regresji liniowej, w którym mamy zestaw danych treningowych. Na podstawie danych treningowych używamy propagacji w przód do przodu do modelowania funkcji przewidywania linii prostej, h (x), jak na poniższym diagramie:




Różnica między rzeczywistą a przewidywaną wartością dla pojedynczej próbki treningowej przyczynia się do ogólnego błędu funkcji prognozowania. Dobroć dopasowania do sieci neuronowej określa się za pomocą funkcji kosztu. Mierzy to, jak dobrze działała sieć neuronowa w odniesieniu do zestawu danych szkoleniowych podczas modelowania danych szkoleniowych. Jak można sobie wyobrazić, wartość funkcji kosztu w przypadku sieci neuronowej zależy od ciężaru każdego neuronu i tendencyjności na każdym z węzłów. Funkcja kosztu jest pojedynczą wartością i jest reprezentatywna dla całej sieci neuronowej. Funkcja kosztów ma następującą postać w sieci neuronowej:

$C(W, X^r, Y^r)$

* W oznacza wagi dla sieci neuronowej

* $X^r$ reprezentuje wartości wejściowe pojedynczej próbki treningowej

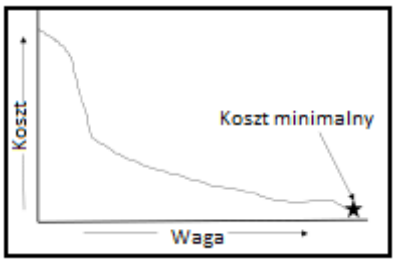
* $Y^r$ reprezentuje wynik odpowiadający $X^r$

Jak widzieliśmy wcześniej, koszt wszystkich punktów danych treningowych można wyrazić jako sumę błędu kwadratu. Dzięki temu otrzymujemy nasze piąte równanie dla sieci neuronowej, które reprezentuje koszt:

$C(W, X^r, Y^r) = J = \Sigma\ 1/2\ (y - y\hat{})^2$

Ponieważ wejściowe dane treningowe są kontekstowe i coś, czego nie możemy kontrolować, celem sieci neuronowej jest wyprowadzenie wag i tendencyjności, aby zminimalizować wartość funkcji kosztu. Gdy minimalizujemy koszty, nasz model dokładniej prognozuje wartości nieznanych danych wejściowych. Istnieje kombinacja wag W, która zapewnia nam minimalny koszt. Patrz rysunek, mamy dziewięć indywidualnych wag w naszej sieci neuronowej. Zasadniczo istnieje kombinacja tych dziewięciu wag, która zapewnia nam minimalny koszt naszego układu nerwowego sieć. Uprośćmy dalej

nasz przykład i załóżmy, że mamy tylko jedną wagę, którą chcemy zoptymalizować, aby zminimalizować koszt hipotezy sieci neuronowej. Możemy zainicjować wagę do wartości losowej i przetestować dużą liczbę dowolnych wartości i wykreślić odpowiedni koszt na prostym dwuwymiarowym wykresie, w następujący sposób:



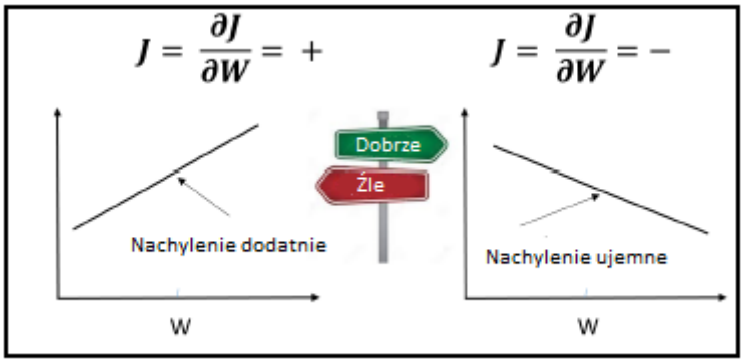

Obliczenie minimalnego kosztu dla dużej liczby wag wejściowych wybranych losowo może być obliczeniowo łatwe i wykonalne. Jednak wraz ze wzrostem liczby wag (w naszym przypadku dziewięciu) wraz z liczbą wymiarów wejściowych (tylko dwa w naszym przykładzie), obliczeniowo niemożliwe jest osiągnięcie minimalnego kosztu w rozsądnym czasie. W rzeczywistych scenariuszach będziemy mieć setki lub tysiące wymiarów i bardzo złożone sieci neuronowe z dużą liczbą ukrytych warstw, a zatem dużą liczbą niezależnych wartości masy. Jak widzimy, metoda optymalizacji brutalnej siły do optymalizacji ciężarów nie będzie działać w przypadku dużej liczby wymiarów. Zamiast tego możemy użyć prostego i szeroko stosowanego algorytmu zejścia gradientowego, aby znacznie zmniejszyć wymagania obliczeniowe podczas szkolenia sieci neuronowej. Aby zrozumieć opadanie gradientu, połączmy nasze pięć równań w jedno równanie w następujący sposób:

$J = \Sigma\ 1/2\ (y - f\ (f\ (X\ W^{(1)})\ W^{(2)}))^2$

W tym przypadku jesteśmy zainteresowani znalezieniem tempa zmian J w stosunku do W, które można przedstawić jako pochodną cząstkową, w następujący sposób:

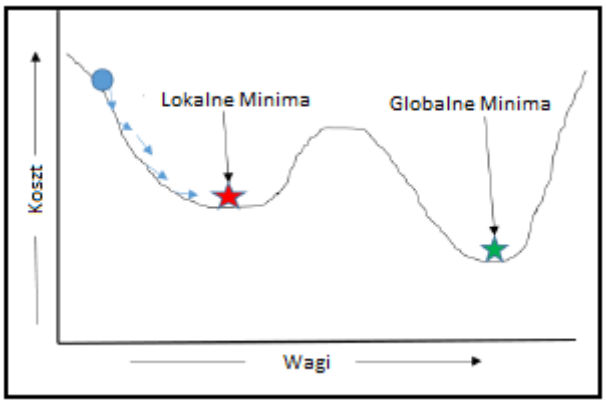
$$J = \frac{\partial J}{\partial W}$$

Jeśli równanie pochodnej ma wartość dodatnią, idziemy pod górę, a nie w kierunku minimalnego kosztu, a jeśli równanie pochodne da wartość ujemną, schodzimy we właściwym kierunku:




Ponieważ znamy kierunek ujemnego nachylenia lub zejście w kierunku obniżonego kosztu sieci neuronowej, możemy zaoszczędzić na kosztach obliczeń idąc w złym kierunku dla kombinacji wartości masy. Możemy iteracyjnie zejść ze wzgórza i zatrzymać się w punkcie, w którym koszt spada do minimum i nie zmienia się znacząco wraz ze zmianą wagi. Sieć neuronowa jest trenowana, gdy otrzymujemy kombinację wag, która daje w wyniku minimalną wartość dla funkcji kosztu. Wraz ze

wzrostem liczby wymiarów i liczby ukrytych warstw wzrasta poziom optymalizacji ze względu na zastosowanie opadania gradientu i możliwe jest wyszkolenie sieci neuronowej. Jednak opadanie gradientu działa dobrze tylko dla relacji funkcji wypukłej między słupami a kosztem. Jeśli związek nie jest wypukły, algorytm spadku gradientu może utknąć w minimach lokalnych zamiast minimów globalnych. Ilustruje to poniższy schemat:




W zależności od tego, w jaki sposób wykorzystujemy nasze dane wejściowe w połączeniu z macierzą wag, może nie mieć znaczenia, czy wykres funkcji kosztu ma charakter niewypukły, jeśli użyjemy przykładów szkolenia i odpowiednich wag pojedynczo w celu przetestowania wielu wartości w kierunku ujemnego nachylenia lub spadku gradientu. Ta technika nazywa się stochastycznym spadkiem gradientu. Wraz ze wzrostem liczby funkcji opadanie gradientu staje się intensywne obliczeniowo i nieuzasadnione w przypadku bardzo złożonych problemów i sieci neuronowych. Stochastyczne opadanie gradientu jest iteracyjną techniką, która może rozdzielić jednostki robocze i doprowadzić nas do globalnych minimów w sposób optymalny obliczeniowo. Aby zrozumieć różnicę między spadkiem gradientu a spadkiem gradientu stochastycznego, spójrzmy na pseudokod dla każdego

**Gradient Descent**

```
for ( i in all_training_examples)
  gradient_descent_params = evaluate_gradient(loss_function, data, parameters)
  parameters = parameters – learning_rate * gradient_descent_params
```

**Stochastic Gradient Descent**

```
for (i in all training_examples)
    random_shuffle(training_data)
        for (single_example in training_data)
            gradient_descent_params = evaluate_gradient(loss_function, single_example, parameters)
            parameters = parameters – learning_rate * gradient_descent_parameters
```

**Pseudokod zejścia gradientu**

Kontynuujemy z pseudokodem opadania gradientu:

1. Niech będzie wartością początkową, którą można wybrać losowo.

2. Oblicz gradient ∂J / ∂W.

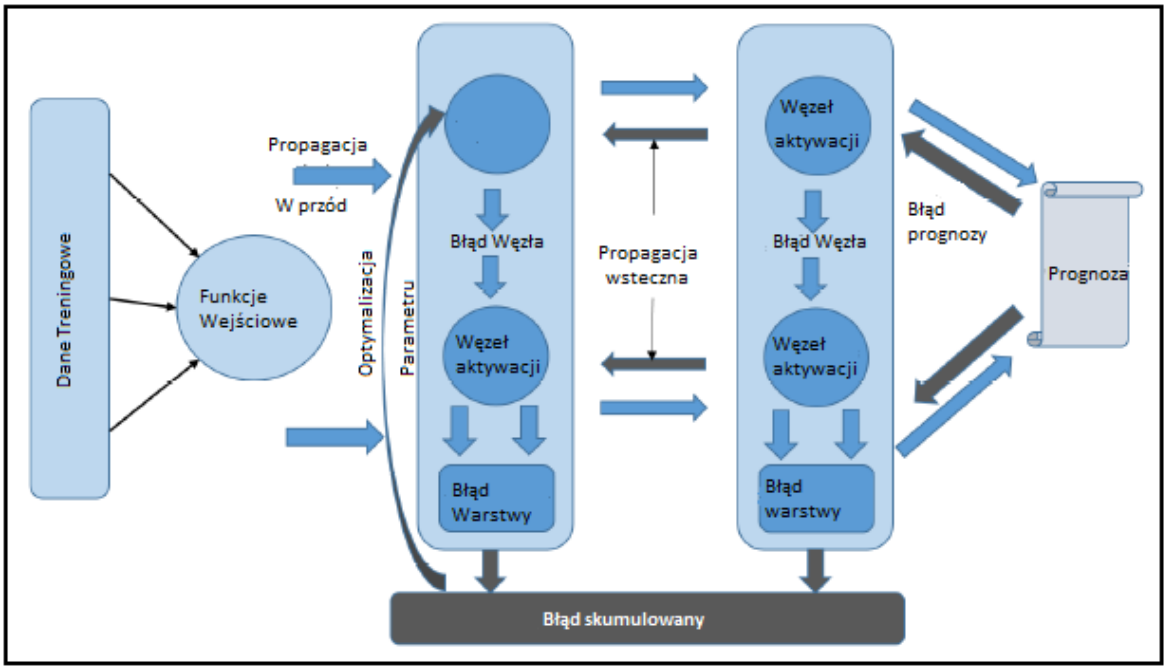
3. Jeśli ∂J / ∂W <t, gdzie t jest pewną z góry określoną wartością progową, WYJŚCIE. Znaleźliśmy wektor wagi, który otrzymuje minimalny błąd dla przewidywanego wyniku.

4. Zaktualizuj W. W = W - s (∂J / ∂W) [s to tak zwana szybkość uczenia się. Należy go starannie wybrać, jeśli jest zbyt duży, gradient zostanie przekroczony, a my przeoczymy minimum. Jeśli jest za duży, zebranie go zajmie zbyt wiele iteracji].

Do tej pory przemierzaliśmy ANN w jednym kierunku, co określa się jako propagację do przodu. Ostatecznym celem szkolenia SSN jest wyprowadzenie ciężarów na każdym z połączeń między węzłami, aby zminimalizować błąd prognozowania. Jedna z najpopularniejszych technik nazywana jest propagacją wsteczną. Podstawową ideą jest to, że gdy poznamy różnicę między rzeczywistą wartością zmiennej predykcyjnej na podstawie przykładu szkolenia, błąd jest obliczany. Błąd w końcowej warstwie wyjściowej jest funkcją wartości aktywacyjnych węzłów na poprzedniej ukrytej warstwie. Każdy węzeł w ukrytej warstwie przyczynia się w różnym stopniu do błędu wyjściowego. Chodzi o to, aby dokładnie wyregulować obciążenia na złączach, aby zminimalizować końcowy błąd wyjściowy. Pomoże nam to zasadniczo określić, jak ukryte jednostki powinny wyglądać na podstawie danych wejściowych i tego, jak mają wyglądać dane wyjściowe. Jest to algorytm online, który otrzymuje dane treningowe, pojedynczo. Posuwamy się naprzód, aby uzyskać prognozy dla klasy przez pomnożenie wag i zastosowanie funkcji aktywacji, uzyskanie błędów prognozowania na podstawie prawdziwej etykiety i przesunięcie błędu z powrotem do sieci w odwrotnym kierunku.

**Model propagacji wstecznej**

Model propagacji wstecznej można przedstawić koncepcyjnie w następujący sposób:




Algorytm propagacji wstecznej można łatwo wdrożyć w sposób etapowy. Jest to mniej wymagające obliczeniowo w porównaniu do spadku gradientu:

* Zainicjuj model: W tym kroku model jest losowo inicjowany do punktu, w którym wagi są wybierane z przybliżeniem matematycznym i losowością. To pierwszy krok w sieci feed-forward.

* Propaguj naprzód: W tym kroku wszystkie jednostki wejściowe, jednostki ukryte i jednostki wyjściowe są aktywowane po dodaniu sumy produktów jednostek neuronowych i wag zaczynając od jednostek wejściowych ze zbiorem danych treningowych. Dane wyjściowe są obliczane przez

zastosowanie aktywacji do końcowej jednostki wyjściowej. Zrozumiałe jest, że wydajność na tym etapie będzie daleka od idealnej oczekiwanej wydajności.

* Obliczanie kosztów: w tym momencie mamy oczekiwany wynik (na podstawie zestawu danych szkoleniowych) i rzeczywisty wynik z nieprzeszkolonej sieci neuronowej. Funkcja kosztu jest zazwyczaj sumą kwadratowych błędów dla każdego punktu danych szkolenia. Jest to matryca wydajności tego, jak dobrze sieć neuronowa pasuje do zbioru danych szkoleniowych, a także wskazanie, jak dobrze jest w stanie uogólnić nieznane wartości wejściowe, które oczekuje się, że model otrzyma po szkoleniu. Pewnego razu funkcja straty została ustalona, celem szkolenia modelu jest zmniejszenie błędu w kolejnych cyklach i dla większości możliwych danych wejściowych, które model napotka w prawdziwym scenariuszu.

* Wyprowadzenie matematyczne funkcji straty: Funkcja straty jest zoptymalizowana za pomocą pochodnej błędu w odniesieniu do wag na każdym z połączeń w sieci neuronowej. Dla każdego z połączeń w sieci neuronowej w tym momencie obliczamy, jaki wpływ ma zmiana wartości pojedynczej masy (w całej sieci) na funkcję strat. Oto niektóre z możliwych scenariuszy, w których obliczamy pochodną kosztu w odniesieniu do wagi:

- Przy określonej wartości masy mamy stratę 0, model dokładnie pasuje do wejściowego zestawu danych treningowych.

- Możemy mieć dodatnią wartość dla funkcji straty, ale pochodna jest ujemna. W tej sytuacji wzrost masy zmniejszy funkcję utraty.

- Możemy mieć dodatnią wartość dla funkcji straty, a pochodna jest również dodatnia. W tej sytuacji zmniejszenie masy ciała spowoduje zmniejszenie funkcji utraty.

* Propagacja wsteczna: na tym etapie błąd w warstwie wyjściowej jest propagowany wstecz do poprzedniej ukrytej warstwy, a następnie z powrotem do warstwy wejściowej. Po drodze obliczamy pochodną i dostosowujemy wagi w podobny sposób, jak w poprzednim kroku. Technika ta nazywana jest automatycznym różnicowaniem w odwrotnym kierunku propagacji do przodu. W każdym węźle obliczamy pochodną straty i dostosowujemy wagę poprzedniego złącza.

* Zaktualizuj wagi: W poprzednim kroku obliczyliśmy pochodne na każdym z węzłów we wszystkich warstwach, propagując ogólny błąd do tyłu. Uproszczony sposób: Nowa waga = Stara waga - (stopa pochodna * stopa uczenia się). Współczynnik uczenia się należy starannie dobierać za pomocą wielu eksperymentów. Jeśli wartość jest zbyt wysoka, możemy przegapić minima, a jeśli wartość jest zbyt niska, model zbiegnie się bardzo powoli. Waga każdego połączenia jest aktualizowana zgodnie z następującymi wytycznymi:

-Jeśli pochodna błędu w stosunku do masy jest dodatnia, wzrost masy proporcjonalnie zwiększy błąd, a nowa waga powinna być mniejsza.

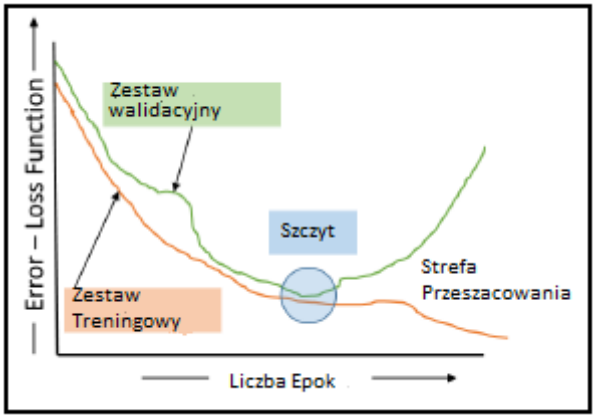
-Jeśli pochodna błędu w stosunku do masy jest ujemna, wzrost masy proporcjonalnie zmniejsza błąd, a nowa waga powinna być większa.

-Jeśli pochodna błędu w stosunku do wagi wynosi 0, nie są wymagane dalsze aktualizacje wag i model sieci neuronowej jest zbieżny.

**Nadmierne dopasowanie**

Jak widzieliśmy w poprzednich sekcjach, opadanie gradientu i propagacja wsteczna są algorytmami iteracyjnymi. Jedno przejście do przodu i odpowiednie przejście do tyłu przez wszystkie dane

treningowe nazywa się epoką. Z każdą epoką model jest szkolony, a wagi są dostosowywane w celu zminimalizowania błędu. W celu przetestowania dokładności modelu, jako powszechną praktykę, podzieliliśmy dane treningowe na zestaw treningowy i zestaw walidacyjny. Zestaw treningowy służy do generowania modelu reprezentującego hipotezę opartą na danych historycznych, które zawierają wartość zmiennej docelowej w odniesieniu do zmiennych niezależnych lub wejściowych. Zestaw walidacyjny służy do testowania wydajności funkcji hipotezy lub wyuczonego modelu dla nowych próbek szkoleniowych. W wielu epokach zwykle obserwujemy następujący wzór:




Gdy trenujemy naszą sieć neuronową przez wiele epok, błąd funkcji utraty jest optymalizowany z każdą epoką, a błąd modelu skumulowanego dąży do 0. W tym momencie model trenował się w odniesieniu do danych treningowych. Kiedy sprawdzamy poprawność hipotezy przy użyciu zestawu sprawdzania poprawności, błąd funkcji utraty zmniejsza się do wartości szczytowej. Po szczycie ponownie błąd zaczyna się zwiększać, jak pokazano na poprzednim rysunku. W tym momencie model zapamiętał dane treningowe i nie jest w stanie uogólnić się dla nowego zestawu danych. Każda epoka po tym punkcie znajduje się w strefie nadmiernego dopasowania. Po tym momencie model przestał się uczyć i przyniesie nieprawidłowe wyniki lub wyniki. Jednym z najprostszych sposobów zapobiegania nadmiernemu dopasowaniu i stworzenia modelu, który dobrze się uogólnia, jest zwiększenie ilości danych treningowych. Wraz ze wzrostem danych treningowych sieć neuronowa jest dostosowywana do coraz większej liczby rzeczywistych scenariuszy, a zatem dobrze się uogólnia. Jednak z każdym wzrostem zestawu danych szkoleniowych koszt obliczeniowy każdej epoki proporcjonalnie wzrasta. Maszyna ma skończoną pojemność do modelowania danych. Zdolność ANN do modelowania można kontrolować, zmieniając liczbę ukrytych jednostek, modyfikując i optymalizując liczbę iteracji treningowych lub zmieniając stopień nieliniowości funkcji aktywacyjnych. Nadmiernym dopasowaniem można kontrolować, zmniejszając liczbę funkcji. Niektóre funkcje mają niewielki wpływ na ogólne zachowanie modelu, a tym samym na wynik. Takie cechy należy zidentyfikować algorytmicznie za pomocą wielu eksperymentów i iteracji i wyeliminować z ostatecznego generowania modelu. Możemy również zastosować techniki regularyzacji, w których wykorzystywane są wszystkie funkcje, ale z różnym stopniem wagi w zależności od znaczenia danej cechy dla ogólnego wyniku. Inną popularną techniką regularyzacji w celu zapobiegania przeuczeniu jest porzucanie. Dzięki tej technice węzły w ANN są ignorowane (upuszczane) podczas fazy szkolenia. Neurony, które są ignorowane, są wybierane losowo.

**Nawracające sieci neuronowe**

Do tej pory widzieliśmy ANN, w których sygnały wejściowe są propagowane do warstwy wyjściowej w przejściu do przodu, a wagi są optymalizowane w sposób rekurencyjny w celu trenowania modelu uogólnienia nowych danych wejściowych na podstawie zestawu treningowego dostarczonego jako

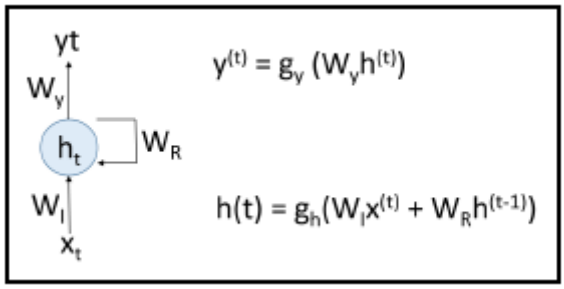
Wejście. Szczególnym problemem w rzeczywistości jest optymalizacja ANN do szkolenia sekwencji danych, na przykład tekstu, mowy lub innej formy wprowadzania dźwięku. Mówiąc najprościej, gdy dane wyjściowe jednej propagacji do przodu są podawane jako dane wejściowe do następnej iteracji treningu, topologia sieci nazywana jest rekurencyjną siecią neuronową (RNN).

**Potrzeba RNN**

W przypadku sieci sprzężenia zwrotnego rozważamy niezależne zestawy danych wejściowych. W przypadku problemów z rozpoznawaniem obrazów mamy obrazy wejściowe, które są od siebie niezależne pod względem zestawu danych wejściowych. W tym przypadku rozważamy macierz pikseli dla obrazu wejściowego. Dane wejściowe dla jednego obrazu nie wpływają na dane wejściowe dla następnego obrazu, który ANN próbuje rozpoznać. Jeśli jednak obraz jest częścią sekwencji lub klatki na wejściu wideo, istnieje korelacja lub zależność między jedną klatką a następną klatką. Dotyczy to również wprowadzania dźwięku lub mowy do ANN. Innym ograniczeniem ANN, które widzieliśmy do tej pory, jest to, że długość warstwy wejściowej musi być stała. Na przykład sieć, która rozpoznaje obraz 27 x 27 pikseli jako dane wejściowe, konsekwentnie będzie w stanie pobierać dane o tym samym rozmiarze do pętli treningowych i uogólniających. RNN może pomieścić wejście o zmiennej długości, a zatem jest bardziej podatne na zmiany sygnałów wejściowych. Podsumowując, RNN są dobre przy wejściach zależnych i wejściowych o zmiennej długości.

**Struktura RNN**

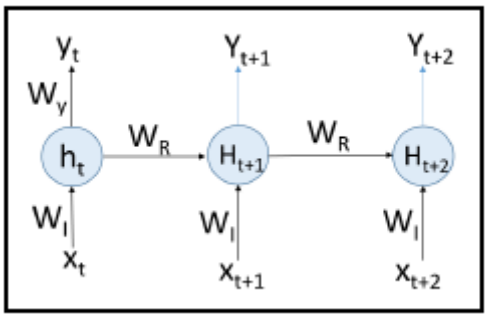
Prosta reprezentacja RNN ma miejsce, gdy weźmiemy pod uwagę wynik jednej iteracji jako dane wejściowe do następnej iteracji propagacji w przód. Można to zilustrować w następujący sposób:




Jednostka liniowa, która odbiera dane wejściowe, $x^t$, stosuje wagę, $W_I$ i generuje hipotezę z metamorfozą funkcji aktywacyjnej do RNN, gdy wprowadzamy macierz wagową WR z powrotem do wyniku funkcji hipotezy w czasie z wprowadzeniem cyklicznego połączenie. W poprzednim przykładzie t reprezentuje aktywację w t przestrzeni czasowej. Teraz aktywność sieci zależy nie tylko od sygnału wejściowego, wagi i funkcji aktywacji, ale także od aktywności poprzedniego znacznika czasu. W formacie równań wszystko jest takie samo, z wyjątkiem wprowadzenia dodatkowego parametru reprezentującego wynik z poprzedniej aktywacji w czasie (t-1).

**Szkolenie RNN**

RNN można wytrenować, rozwijając jednostkę cykliczną w odpowiednim czasie w szeregu sieci sprzężenia zwrotnego:




Najbardziej lewą jednostką jest aktywność sieci w czasie, t, która jest typową siecią sprzężenia zwrotnego z $x_t$ jako wejściem w czasie, t. Mnoży się to przez macierz masy, WI. Dzięki zastosowaniu funkcji aktywacji otrzymujemy dane wyjściowe, $y_t$, w czasie, t. Wyjście to jest podawane jako dane wejściowe do następnej jednostki wraz z kontekstowym i czasowym wprowadzaniem następnej jednostki w czasie, t + 1. Jeśli zauważysz, istnieje zasadnicza różnica w sieci feed-forward i RNN. Wagi w różnych warstwach wejściowych, ukrytych i wyjściowych w sieci sprzężenia zwrotnego różnią się od siebie i reprezentują znaczenie zmiennej zależnej oraz połączenia na całej mocy wyjściowej. W przypadku RNN wagi w jednostkach (WR), które są rozwijane w czasie, są takie same. Ponieważ będziemy mieli wynik w każdej jednostce, będziemy mieli koszt związany z każdą jednostką. Załóżmy, że koszt pierwszej jednostki w znaczniku czasu t to $C_t$, a kolejne jednostki jako $C_{t+1}$ i $C_{t+2}$. Szkolenie RNN można przedstawić matematycznie jako:

$$\frac{\partial C}{\partial W_R} = \sum_t \frac{\partial C_t}{\partial W_R} \Rightarrow \frac{\partial C_2}{\partial W_R} = \frac{\partial C_2}{\partial y_2}\frac{\partial y_2}{\partial h_2}\frac{\partial h_2}{\partial g}\frac{\partial g}{\partial a}\frac{\partial a}{\partial W_R}$$

W tym przypadku łączymy gradienty między jednostkami, aby obliczyć całkowity koszt sieci. Ponieważ wagi są dzielone między jednostki, funkcja kosztu jest pochodną w odniesieniu do wag i możemy to wywnioskować za pomocą tych samych metod propagacji wstecznej i gradientu spadku. Po przeszkoleniu RNN można go używać przede wszystkim w scenariuszach, w których dane wejściowe są od siebie zależne. W przypadku tłumaczenia językowego możemy użyć połączeń między dwoma słowami kluczowymi, aby przewidzieć następne słowo w sekwencji, aby zwiększyć dokładność modelu tłumaczenia językowego.

**Często Zadawane Pytania**

P: Czy ANN są dokładnie takie same jak neurony biologiczne pod względem przechowywania i przetwarzania informacji?

O: Chociaż nie można stwierdzić ze 100% pewnością, że ANN są dokładną repliką pod względem pamięci i logiki przetwarzania, w naukach medycznych istnieją dowody, że podstawowym budulcem mózgu jest neuron, a neurony są ze sobą połączone. Po uzyskaniu zewnętrznego bodźca lub gdy jest on generowany przez procesy mimowolne, neurony reagują komunikując się ze sobą poprzez przekazywanie sygnałów neuronalnych. Chociaż funkcjonowanie mózgu jest bardzo skomplikowane i dalekie od pełnego zrozumienia, teoria SSN ewoluuje i widzimy wiele sukcesów w modelowaniu niektórych bardzo złożonych problemów, które nie byłyby możliwe w przypadku tradycyjnych modeli programowania. Aby stworzyć nowoczesne maszyny posiadające zdolności poznawcze ludzkiego mózgu, potrzebne są dalsze badania i znacznie lepsze zrozumienie biologicznych sieci neuronowych.

P: Jakie są podstawowe elementy składowe ANN?

O: ANN składa się z różnych warstw. Warstwa, która otrzymuje dane wejściowe ze środowiska (zmienne niezależne), jest zużywana przez warstwę wejściową. Istnieje ostatnia warstwa, która emituje dane wyjściowe modelu na podstawie uogólnienia danych treningowych. Ta warstwa jest nazywana warstwą wyjściową. Pomiędzy warstwami wejściową i wyjściową może znajdować się jedna lub wiele warstw przetwarzających sygnały. Te warstwy nazywane są warstwami ukrytymi. Węzły w każdej z warstw są połączone streszczeniem lub łącznikami. Każde złącze ma optymalną wagę, aby zmniejszyć wartość funkcji kosztu, która reprezentuje dokładność sieci neuronowej.

P: Jaka jest potrzeba nieliniowości w ramach ANN?

O: Sieci neuronowe są modelami matematycznymi, w których dane wejściowe są mnożone przez masy synoptyczne, a suma wszystkich produktów połączenia węzła stanowi wartość w węźle. Jeśli jednak nie włączymy nieliniowości z funkcją aktywacji, wielowarstwowe sieci neuronowe nie będą istnieć. W takim przypadku model można przedstawić za pomocą jednej ukrytej warstwy. Będziemy mogli modelować bardzo proste problemy z modelowaniem liniowym. Aby modelować bardziej złożone problemy w świecie rzeczywistym, potrzebujemy wielu warstw i tym samym nieliniowość funkcji aktywacyjnych.

P: Jakie funkcje aktywacyjne są najczęściej używane przy tworzeniu ANN?

O: Często stosowanymi funkcjami aktywacyjnymi w ramach ANN są: Funkcja sigmoidalna: Wartość wyjściowa wynosi od 0 do 1. Ta funkcja przyjmuje kształt geometryczny S, a zatem nazwa sigmoid. Funkcja Tanh: Funkcja stycznej hiperbolicznej (tanh) jest niewielką odmianą funkcji sigmoidalnej, która jest wyśrodkowana na 0. Rectified Linear Unit (ReLu): Jest to najprostsza, zoptymalizowana obliczeniowo, a zatem najczęściej używana funkcja aktywacji dla ANN. Wartość wyjściowa wynosi 0 dla wszystkich wejść ujemnych i jest taka sama jak wartość wejściowa dla wejść dodatnich.

P: Co to jest ANN z informacją zwrotną i jak wybiera się początkowe wartości wag?

O: Pojedyncze przejście przez sieć od warstwy wejściowej do warstwy wyjściowej przez warstwy ukryte nazywa się przejściem do przodu. Podczas tego węzły są aktywowane jako suma iloczynów wartości węzłów i wag połączeń. Początkowe wartości wag są wybierane losowo, w wyniku czego wynik pierwszego przejścia może różnić się od oczekiwanego wyniku na podstawie danych treningowych. Ta delta nazywana jest kosztem sieci i jest reprezentowana przez funkcję kosztu. Intuicja i celem SSN jest ostateczne ograniczenie kosztów do minimum. Osiąga się to poprzez wielokrotne przejścia przez sieć do przodu i do tyłu. Jedna podróż w obie strony nazywa się epoką.

P: Jakie jest znaczenie nadmiernego dopasowania modelu?

O: Nadmierne dopasowanie modelu występuje, gdy model uczy się danych wejściowych i nie może generalizować nowych danych wejściowych. Gdy to nastąpi, model praktycznie nie nadaje się do rozwiązywania problemów w świecie rzeczywistym. Przeregulowanie można zidentyfikować na podstawie różnic w dokładności modelu między przebiegami szkolenia i zestawami danych do walidacji.

P: Co to są RNN i gdzie są używane?

O: RNN to rekurencyjne sieci neuronowe, które wykorzystują dane wyjściowe z jednego przejścia do przodu przez sieć jako dane wejściowe do następnej iteracji. RNN są używane, gdy dane wejściowe nie są od siebie niezależne. Na przykład model tłumaczenia języka musi przewidywać następne możliwe słowo na podstawie poprzedniej sekwencji słów. ANN mają ogromne znaczenie w dziedzinie przetwarzania języka naturalnego i systemów przetwarzania audio / wideo.

Podsumowanie

Przedstawiliśmy najważniejszą koncepcję realizacji inteligentnych maszyn, którymi są sztuczne sieci neuronowe. ANN są modelowane w stosunku do mózgu biologicznego. Chociaż teoria ANN istniała od dziesięcioleci, pojawienie się rozproszonej mocy obliczeniowej wraz z dostępem do niespotykanej ilości danych umożliwiło rozwój w tej ekscytującej dziedzinie badań. W tym rozdziale przedstawiliśmy podstawowe elementy składowe ANN i proste techniki trenowania modeli w celu uogólnienia modelu generowania wyników dla nowych zestawów danych. To wprowadzenie stanowi element składowy następnego rozdziału, który zagłębi się głębiej w aspekty implementacji sieci neuronowych.