Budowanie sieci neuronowej w Pythonie

Sieci neuronowe i różne inne modele działania mózgu istniały tak długo, jak ludzie mówili o sztucznej inteligencji. Marvin Minsky, którego przedstawiliśmy wczesniej, rozpoczął główny nurt zainteresowania modelowaniem neuronów swoją przełomową pracą nad perceptronami w 1969 roku. W tamtym czasie panowała powszechna „irracjonalna euforia" dotycząca tego, jak perceptron bardzo szybko uczyni sztuczną inteligencję praktyczną. To przyciągnęło sporo kapitału wysokiego ryzyka do tego obszaru, ale kiedy wiele z tych przedsięwzięć zakończyło się niepowodzeniem, inwestycje w sieci neuronowe wyschły. To jest jak koncepcja mody w nauce. To, co jest popularne, sprzedaje się. Szybko do przodu o 30 lat. Obecnie obserwuje się ponowne zainteresowanie sieciami neuronowymi. Zbudowano lepsze modele, ale naprawdę ważne jest to, że mamy teraz realne, użyteczne i ekonomiczne aplikacje oparte na sieciach neuronowych. Czy to doprowadzi do kolejnej bańki? Z całą pewnością, ale tym razem zainteresowanie powinno być kontynuowane ze względu na obszary zastosowań, które zostały opracowane. W tej części zapoznasz się z koncepcją sieci neuronowych i sposobami ich implementacji w Pythonie.

Zrozumienie sieci neuronowych

Oto sześć atrybutów sieci neuronowej:

* Warstwa wejściowa neuronów

* Dowolna ilość ukrytych warstw neuronów

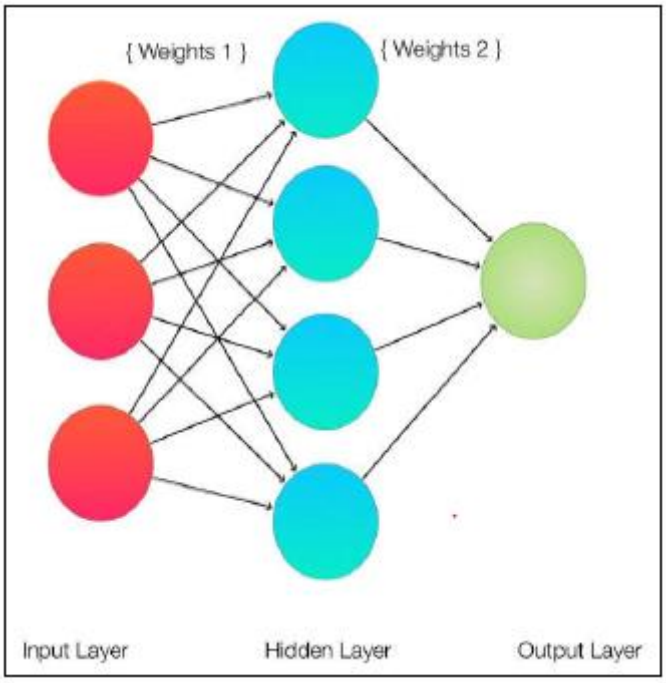
* Warstwa wyjściowa neuronów łączących się ze światem

* Zestaw wag i odchyleń między każdym poziomem neuronu

* Wybór funkcji aktywacji dla każdej ukrytej warstwy neuronów

* Funkcja strat, która zapewni „przetrenowanie" sieci

Rysunek przedstawia architekturę dwuwarstwowej sieci neuronowej.



Zwróć uwagę na trzy warstwy w tej „dwuwarstwowej” sieci neuronowej: Warstwa wejściowa jest zwykle pomijana podczas liczenia warstw sieci neuronowej. Patrząc na ten diagram, można zobaczyć, że neurony na każdej warstwie są połączone ze wszystkimi neuronami następnej warstwy. Wagi podano dla każdej z linii połączeń między neuronami. Neuron, jak to słowo jest używane w AI, to programowy model komórki układu nerwowego, który zachowuje się mniej więcej jak prawdziwy neuron mózgowy. Model wykorzystuje liczby, aby jeden neuron był ważniejszy dla wyników. Liczby te nazywane są wagami.

Warstwy neuronów

Rysunek powyżej przedstawia warstwę wejściową, warstwę ukrytą (tak zwaną, ponieważ nie jest bezpośrednio połączona ze światem zewnętrznym) oraz warstwę wyjściową. To jest bardzo prosta sieć; rzeczywiste sieci mogą być znacznie bardziej złożone z wieloma dodatkowymi warstwami. W rzeczywistości głębokie uczenie bierze swoją nazwę od faktu, że masz wiele ukrytych warstw, co w pewnym sensie zwiększa „głębokość” sieci neuronowej. Zwróć uwagę, że warstwy są filtrowane i przetwarzają informacje od lewej do prawej w sposób progresywny. Nazywa się to wejściem sprzężenia zwrotnego, ponieważ dane są podawane tylko w jednym kierunku. Więc teraz, kiedy mamy sieć, jak ona się uczy? Sieć neuronowa otrzymuje przykład i zgaduje odpowiedź (używając dowolnych domyślnych wag i odchyleń, od których zaczynają). Jeśli odpowiedź jest błędna, cofa się i modyfikuje wagi i uprzedzenia w neuronach i próbuje naprawić błąd, zmieniając niektóre wartości. Nazywa się to propagacją wsteczną i symuluje to, co ludzie robią podczas wykonywania zadania, stosując iteracyjne podejście metodą prób i błędów. Po wykonaniu tego procesu wiele razy, w końcu sieć neuronowa zaczyna się poprawiać (uczy się) i dostarcza lepszych odpowiedzi. Każda z tych iteracji nazywana jest epoką. Ta nazwa pasuje całkiem dobrze, ponieważ czasami szkolenie w celu nauczenia się złożonych zadań może zająć dni lub tygodnie. W tym momencie należy zwrócić uwagę na fakt, że chociaż nauczenie sieci neuronowej może zająć kilka dni lub tygodni, po jej wytrenowaniu możemy ją powielić przy niewielkim wysiłku, kopiując topologię, wagi i odchylenia wytrenowanej sieci. Kiedy masz wytrenowaną sieć neuronową, możesz z łatwością używać jej wielokrotnie, dopóki nie będziesz potrzebować czegoś innego. Potem wracamy do treningów. Sieci neuronowe jako takie modelują niektóre rodzaje ludzkiego uczenia się. Jednak ludzie dysponują znacznie bardziej złożonymi sposobami hierarchicznego kategoryzowania obiektów (takich jak kategoryzowanie koni i świń jako zwierząt) przy niewielkim wysiłku. Sieci neuronowe (i cała dziedzina głębokiego uczenia) nie są zbyt dobre w przekazywaniu wiedzy i wyników z jednego rodzaju sytuacji do drugiego bez ponownego szkolenia.

Wagi i odchylenia

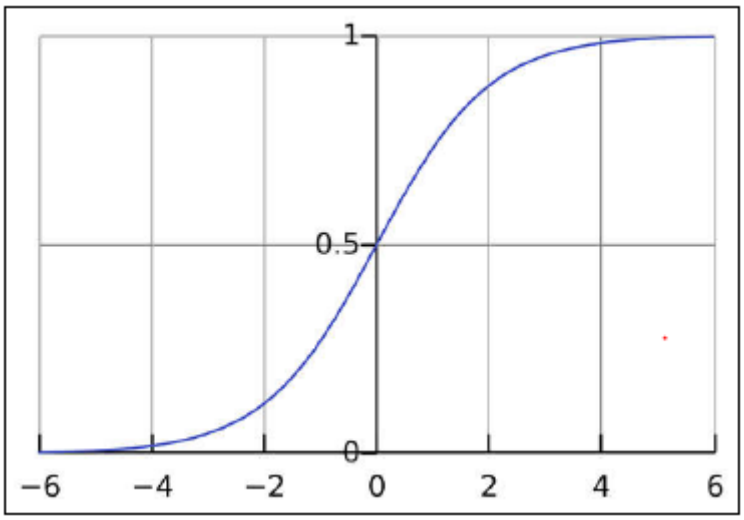
Patrząc na sieć na rysunku powyżej, widać, że wydajność tych sieci neuronowych zależy tylko od wag połączeń wzajemnych, a także od czegoś, co nazywamy odchyleniami samych neuronów. Chociaż wagi wpływają na stromość krzywej funkcji aktywacji (więcej o tym później), odchylenie przesunie całą krzywą w prawo lub w lewo. Wybór wag i odchyleń określa siłę przewidywań poszczególnych neuronów. Uczenie sieci neuronowej polega na wykorzystaniu danych wejściowych do precyzyjnego dostrojenia wag i odchyleń.

WSTECZNA PROPAGACJA

W ludzkim mózgu uczenie się odbywa się dzięki dodawaniu nowych połączeń (synaps) i modyfikacji tych połączeń w oparciu o bodźce zewnętrzne. Metody stosowane do propagacji wyników z poprzednich warstw (zwane również sprzężeniem zwrotnym) zmieniły się na przestrzeni lat w badaniach AI, a niektórzy eksperci twierdzą, że za ostatnim wzrostem aplikacji AI i wyjściem z ostatniej „Zimy AI” stoi zmiana algorytmów i technik wykorzystywanych do propagacji wstecznej. Propagacja wsteczna jest dość złożonym matematycznie tematem

Funkcja aktywacji

Funkcja aktywacji jest ważnym tematem do omówienia podczas budowania naszej pierwszej sieci neuronowej. Jest to kluczowa część naszego modelu neuronów. Jest to funkcja oprogramowania, która określa, czy informacja przechodzi przez pojedynczy neuron, czy też jest przez niego zatrzymywana. Jednak nie używasz go tylko jako bramki (otwartej lub zamkniętej), używasz go jako funkcji, która przekształca sygnał wejściowy do neuronu w jakiś użyteczny sposób. Dostępnych jest wiele rodzajów funkcji aktywacji. W przypadku naszej prostej sieci neuronowej wykorzystamy jedną z najpopularniejszych - funkcję sigmoidalną. Funkcja sigmoidalna ma charakterystyczną krzywą „S", jak pokazano na rysunku



Pamiętasz, jak mówiliśmy o stronniczości neuronów we wcześniejszej części? Jeśli do krzywej z rysunku zastosujesz odchylenie o wartości 1,0, cała krzywa przesunie się w prawo, powodując przesunięcie punktu (0,0,5) do punktu (1,0,5).

Funkcja straty

Funkcja straty jest ostatnim elementem układanki, który należy wyjaśnić. Funkcja straty porównuje wynik naszej sieci neuronowej z pożądanymi wynikami. Innym sposobem myślenia o tym jest to, że funkcja straty mówi nam, jak dobre są nasze obecne wyniki. To jest informacja, której szukamy, aby dostarczyć ją do naszego kanału wstecznej propagacji, który poprawi naszą sieć neuronową. Zamierzam użyć funkcji, która wyznacza pochodną funkcji straty w stosunku do naszego wyniku (nachylenie krzywej to pierwsza pochodna wachlarzy rachunku różniczkowego), aby dowiedzieć się, co zrobić z masami naszych neuronów. Jest to główna część „uczenia się" działalności sieci.

X1	X2	X3	Y1
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Funkcja Exclusive Or zwraca wartość 1 tylko wtedy, gdy wszystkie dane wejściowe mają wartość 0 lub 1.

Kod Pythona sieci neuronowej

Będziemy korzystać z biblioteki Pythona o nazwie NumPy, która zapewnia świetny zestaw funkcji pomagających nam zorganizować naszą sieć neuronową, a także upraszcza obliczenia.

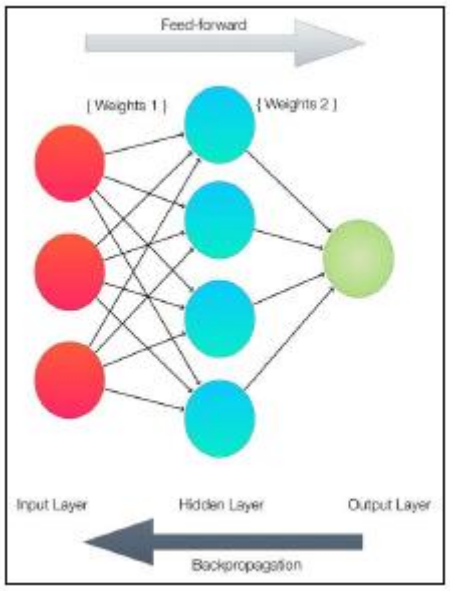
ZASTOSOWANIA FUNKCJI XOR

Bramki XOR są używane w wielu różnych aplikacjach, zarówno programowych, jak i sprzętowych. Możesz użyć bramki XOR jako części sumatora jednobitowego, który dodaje jeden bit do innego bitu (i zapewnia bit przeniesienia, aby połączyć je razem w celu utworzenia dużych sumatorów), a także połączyć je razem, aby zbudować liczbę pseudolosową generator. Najfajniejsze znane nam zastosowanie bramki XOR dotyczy algorytmów kodowania i algorytmu korekcji błędu Reeda-Solomonerrora. Algorytmy Reeda-Solomona w pewnym sensie mieszają twoje dane, używając bramek XOR, a następnie dodając dodatkowe dane (dane nadmiarowe - rodzaj połączenia twoich danych), a następnie masz bardziej niezawodne dane do przesyłania na duże odległości (jak z Plutona, nasza dawna dziewiąta planeta), na której mogą wystąpić różnego rodzaju zdarzenia powodujące szum w danych, uszkadzając bity i bajty. Kiedy otrzymujesz dane, ponownie używasz bramek XOR, aby zrekonstruować oryginalne dane, poprawiając wszelkie błędy (do pewnego momentu), aby uzyskać dobre dane. To pozwala nam przesyłać dane znacznie dalej przy mniejszej mocy, ponieważ dzięki kodowi Reed-Solomon stajesz się odporny na błędy. Dlaczego nic o tym nie wiemy? Ponieważ John pracował z zespołem nad chipami do kodu Reed-Solomon od lat 80. na Uniwersytecie Idah lub NASA. Nasze pochodne piasku z wiórów trafiły do projektów takich jak Kosmiczny Teleskop Hubble'a i osobisty ulubieniec Johna, sonda kosmiczna New Horizons, która odwiedziła Plutona, a ostatnio odwiedziła Ultima Thule w Obłoku Oorta. Wszystkie niesamowite zdjęcia przechodzą przez te wszystkie małe bramki XOR.

NumPy - NUMERYCZNY PYTHON

NumPy to biblioteka Pythona zaprojektowana w celu uproszczenia pisania kodu dla matematyki macierzowej (macierz jest również znana jako tensor) w algebrze liniowej. NumPy zawiera również szereg wyższych funkcji matematycznych, które są przydatne w różnych typach sztucznej inteligencji. Początek rozwoju NumPy sięga 1995 roku i jednego z oryginalnych pakietów algebry macierzowej Pythona. Jest to obecnie preferowana biblioteka, a także część SciPy i MatPlotLib, dwóch popularnych pakietów naukowych do analizy i wizualizacji danych.

Poniżej znajduje się nasz kod Pythona wykorzystujący NumPy dla dwuwarstwowej sieci neuronowej z rysunku .



Używając nano (lub swojego ulubionego edytora tekstu), otwórz plik o nazwie „2Layer-NeuralNetwork.py” i wprowadź następujący kod:

```
# 2 Layer Neural Network in NumPy

import numpy as np

# X = input of our 3 input XOR gate

# set up the inputs of the neural network (right from the table)

X = np.array(([0,0,0],[0,0,1],[0,1,0], \

[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)

# y = our output of our neural network

y = np.array(([1], [0], [0], [0], [0], \

[0], [0], [1]), dtype=float)

# what value we want to predict

xPredicted = np.array(([0,0,1]), dtype=float)

X = X/np.amax(X, axis=0) # maximum of X input array

# maximum of xPredicted (our input data for the prediction)

xPredicted = xPredicted/np.amax(xPredicted, axis=0)

# set up our Loss file for graphing

lossFile = open("SumSquaredLossList.csv", "w")

class Neural_Network (object):
```

```
def __init__(self):
#parameters
self.inputLayerSize = 3 # X1,X2,X3
self.outputLayerSize = 1 # Y1
self.hiddenLayerSize = 4 # Size of the hidden layer
# build weights of each layer
# set to random values
# look at the interconnection diagram to make sense of this
# 3x4 matrix for input to hidden
self.W1 = \
np.random.randn(self.inputLayerSize, self.hiddenLayerSize)
# 4x1 matrix for hidden layer to output
self.W2 = \
np.random.randn(self.hiddenLayerSize, self.outputLayerSize)
def feedForward(self, X):
# feedForward propagation through our network
# dot product of X (input) and first set of 3x4 weights
self.z = np.dot(X, self.W1)
# the activationSigmoid activation function - neural magic
self.z2 = self.activationSigmoid(self.z)
# dot product of hidden layer (z2) and second set of 4x1 weights
self.z3 = np.dot(self.z2, self.W2)
# final activation function - more neural magic
o = self.activationSigmoid(self.z3)
return o
def backwardPropagate(self, X, y, o):
# backward propagate through the network
# calculate the error in output
self.o_error = y - o
# apply derivative of activationSigmoid to error
self.o_delta = self.o_error*self.activationSigmoidPrime(o)
```

```
# z2 error: how much our hidden layer weights contributed to output
# error
self.z2_error = self.o_delta.dot(self.W2.T)
# applying derivative of activationSigmoid to z2 error
self.z2_delta = self.z2_error*self.activationSigmoidPrime(self.z2)
# adjusting first set (inputLayer --> hiddenLayer) weights
self.W1 = X.T.dot(self.z2_delta)
# adjusting second set (hiddenLayer --> outputLayer) weights
self.W2 = self.z2.T.dot(self.o_delta)
def trainNetwork(self, X, y):
# feed forward the loop
o = self.feedForward(X)
# and then back propagate the values (feedback)
self.backwardPropagate(X, y, o)
def activationSigmoid(self, s):
# activation function
# simple activationSigmoid curve as in the book
return 1/(1 np.exp(-s))
```

```
def activationSigmoidPrime(self, s):
# First derivative of activationSigmoid
# calculus time!
return s * (1 - s)
def saveSumSquaredLossList(self,i,error):
lossFile.write(str(i) "," str(error.tolist()) '\n')
def saveWeights(self):
# save this in order to reproduce our cool network
```

```
np.savetxt("weightsLayer1.txt", self.W1, fmt="%s")

np.savetxt("weightsLayer2.txt", self.W2, fmt="%s")

def predictOutput(self):

print ("Predicted XOR output data based on trained weights: ")

print ("Expected (X1-X3): \n" + str(xPredicted))

print ("Output (Y1): \n" + str(self.feedForward(xPredicted)))

myNeuralNetwork = Neural_Network()

trainingEpochs = 1000

#trainingEpochs = 100000

for i in range(trainingEpochs): # train myNeuralNetwork 1,000 times

print ("Epoch # " + str(i) + "\n")

print ("Network Input : \n" + str(X))

print ("Expected Output of XOR Gate Neural Network: \n" + str(y))

print ("Actual Output from XOR Gate Neural Network: \n " + \

str(myNeuralNetwork.feedForward(X)))

# mean sum squared loss

Loss = np.mean(np.square(y - myNeuralNetwork.feedForward(X)))

myNeuralNetwork.saveSumSquaredLossList(i,Loss)

print ("Sum Squared Loss: \n" + str(Loss))

print ("\n")

myNeuralNetwork.trainNetwork(X, y)

myNeuralNetwork.saveWeights()

myNeuralNetwork.predictOutput()
```

Łamanie kodu

Niektóre z poniższych kodów są nieco zagmatwane za pierwszym razem, więc podamy kilka wyjaśnień.

```
# 2 Layer Neural Network in NumPy

import numpy as np.
```

Jeśli podczas uruchamiania powyższego kodu wystąpi błąd importu, zainstaluj bibliotekę NumPy Python. Aby to zrobić na Raspberry Pi (lub systemie Ubuntu), wpisz następujące polecenie w oknie terminala:

```
sudo apt-get install python3-numpy
```

Następnie określamy wszystkie osiem możliwości naszych wejść X1–X3 i wyjścia Y1 z tabeli 1.

```
# X = input of our 3 input XOR gate

# set up the inputs of the neural network (right from the table)

X = np.array(([0,0,0],[0,0,1],[0,1,0], \

[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)

# y = our output of our neural network

y = np.array(([1], [0], [0], [0], [0], \

[0], [0], [1]), dtype=float)
```

Wybieramy wartość do przewidzenia (przewidujemy je wszystkie, ale to jest konkretna odpowiedź, której chcemy na końcu).

```
# what value we want to predict

xPredicted = np.array(([0,0,1]), dtype=float)

X = X/np.amax(X, axis=0) # maximum of X input array

# maximum of xPredicted (our input data for the prediction)

xPredicted = xPredicted/np.amax(xPredicted, axis=0)
```

Zapisz nasze wyniki Sum Squared Loss w pliku do wykorzystania przez program Excel na epokę.

```
# set up our Loss file for graphing

lossFile = open("SumSquaredLossList.csv", "w")
```

Zbuduj klasę Neural_Network dla naszego problemu. Rysunek 2 przedstawia sieć, którą budujemy. Możesz zobaczyć, że każda z warstw jest reprezentowana przez linię w sieci.

```
class Neural_Network (object):

def __init__(self):

#parameters

self.inputLayerSize = 3 # X1,X2,X3

self.outputLayerSize = 1 # Y1

self.hiddenLayerSize = 4 # Size of the hidden layer
```

Aby rozpocząć, ustaw wszystkie wagi sieci na losowe wartości.

```
# build weights of each layer

# set to random values

# look at the interconnection diagram to make sense of this

# 3x4 matrix for input to hidden

self.W1 = \

np.random.randn(self.inputLayerSize, self.hiddenLayerSize)
```

```
# 4x1 matrix for hidden layer to output

self.W2 = \

np.random.randn(self.hiddenLayerSize, self.outputLayerSize)
```

Nasza funkcja feedForward implementuje ścieżkę sprzężenia zwrotnego przez sieć neuronową. Zasadniczo mnoży to macierze zawierające wagi z każdej warstwy do każdej warstwy, a następnie stosuje funkcję aktywacji sigmoidalnej.

```
def feedForward(self, X):

# feedForward propagation through our network

# dot product of X (input) and first set of 3x4 weights

self.z = np.dot(X, self.W1)

# the activationSigmoid activation function - neural magic

self.z2 = self.activationSigmoid(self.z)

# dot product of hidden layer (z2) and second set of 4x1 weights

self.z3 = np.dot(self.z2, self.W2)

# final activation function - more neural magic

o = self.activationSigmoid(self.z3)

return o
```

A teraz dodajemy funkcję „backupPropagate”, która implementuje prawdziwe uczenie metodą prób i błędów, z której korzysta nasza sieć neuronowa.

```
def backwardPropagate(self, X, y, o):

# backward propagate through the network

# calculate the error in output

self.o_error = y - o

# apply derivative of activationSigmoid to error

self.o_delta = self.o_error*self.activationSigmoidPrime(o)

# z2 error: how much our hidden layer weights contributed to output

# error

self.z2_error = self.o_delta.dot(self.W2.T)

# applying derivative of activationSigmoid to z2 error

self.z2_delta = self.z2_error*self.activationSigmoidPrime(self.z2)

# adjusting first set (inputLayer --> hiddenLayer) weights

self.W1 = X.T.dot(self.z2_delta)
```

adjusting second set (hiddenLayer --> outputLayer) weights

self.W2 = self.z2.T.dot(self.o_delta)

Aby wytrenować sieć dla określonej epoki, za każdym razem, gdy trenujemy sieć, wywołujemy zarówno funkcje BackwardPropagate, jak i feedForward.

def trainNetwork(self, X, y):

feed forward the loop

o = self.feedForward(X)

and then back propagate the values (feedback)

self.backwardPropagate(X, y, o)

Poniżej przedstawiono funkcję aktywacji sigmoidy i pierwszą pochodną funkcji aktywacji sigmoidy.

def activationSigmoid(self, s):

activation function

simple activationSigmoid curve as in the book

return 1/(1 np.exp(-s))

def activationSigmoidPrime(self, s):

First derivative of activationSigmoid

calculus time!

return s * (1 - s)

Następnie zapisz wartości epoki funkcji straty w pliku dla programu Excel i wagi neuronowe.

def saveSumSquaredLossList(self,i,error):

lossFile.write(str(i) "," str(error.tolist()) '\n')

def saveWeights(self):

save this in order to reproduce our cool network

np.savetxt("weightsLayer1.txt", self.W1, fmt="%s")

np.savetxt("weightsLayer2.txt", self.W2, fmt="%s")

Następnie uruchamiamy naszą sieć neuronową, aby przewidzieć wyniki na podstawie aktualnie wytrenowanych wag.

def predictOutput(self):

print ("Predicted XOR output data based on trained weights: ")

print ("Expected (X1-X3): \n" str(xPredicted))

print ("Output (Y1): \n" str(self.feedForward(xPredicted)))

myNeuralNetwork = Neural_Network()

trainingEpochs = 1000

#trainingEpochs = 100000

Poniżej znajduje się główna pętla treningowa, która obejmuje wszystkie żądane epoki. Zmień powyższą zmienną trainingEpochs, aby zmienić liczbę epok, w których chcesz trenować swoją sieć.

for i in range(trainingEpochs): # train myNeuralNetwork 1,000 times

print ("Epoch # " str(i) "\n")

print ("Network Input : \n" str(X))

print ("Expected Output of XOR Gate Neural Network: \n" str(y))

print ("Actual Output from XOR Gate Neural Network: \n" \

str(myNeuralNetwork.feedForward(X)))

mean sum squared loss

Loss = np.mean(np.square(y - myNeuralNetwork.feedForward(X)))

myNeuralNetwork.saveSumSquaredLossList(i,Loss)

print ("Sum Squared Loss: \n" str(Loss))

print ("\n")

myNeuralNetwork.trainNetwork(X, y)

Zapisz wyniki swojego szkolenia do ponownego wykorzystania i przewiduj wynik naszej żądanej wartości.

myNeuralNetwork.saveWeights()

myNeuralNetwork.predictOutput()

Uruchamianie kodu sieci neuronowej

W wierszu polecenia wprowadź następujące polecenie:

python3 2LayerNeuralNetworkCode.py

Zobaczysz, jak program zaczyna przechodzić przez 1000 epok szkolenia, drukuje wyniki każdej epoki, a następnie pokazuje końcowe dane wejściowe i wyjściowe. Tworzy również następujące interesujące pliki:

* weightsLayer1.txt: Ten plik zawiera

końcowe wytrenowane wagi dla połączeń między warstwą wejściową a warstwą ukrytą (macierz 4x3).

* weightsLayer2.txt: Ten plik zawiera ostateczne wytrenowane wagi dla ukrytych połączeń między warstwami wyjściowymi (macierz 1x4).

* SumSquaredLossList.csv: To jest rozdzielane przecinkami

plik zawierający numer epoki i każdy współczynnik strat na końcu każdej epoki. Używamy tego do wykreślenia wyników we wszystkich epokach

Oto końcowe wyjście programu dla ostatniej epoki (999, ponieważ zaczynamy od 0).

Epoch # 999

Network Input :

[[0. 0. 0.]

[0. 0. 1.]

[0. 1. 0.]

[0. 1. 1.]

[1. 0. 0.]

[1. 0. 1.]

[1. 1. 0.]

[1. 1. 1.]]

Expected Output of XOR Gate Neural Network:

[[1.]

[0.]

[0.]

[0.]

[0.]

[0.]

[0.]

[1.]]

Actual Output from XOR Gate Neural Network:

[[0.93419893]

[0.04425737]

[0.01636304]

[0.03906686]

[0.04377351]

[0.01744497]

[0.0391143]

[0.93197489]]

Sum Squared Loss:

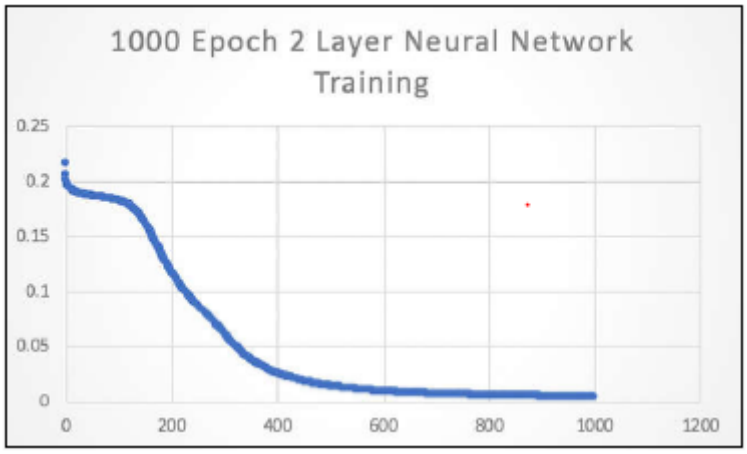
0.0020575319565093496

Predicted XOR output data based on trained weights:

Expected (X1-X3):

[0. 0. 1.]

Na dole widać, że nasz oczekiwany wynik to 0. 04422615, co jest dość bliskie, ale niezupełnie oczekiwanej wartości 0. Jeśli porównasz każdy z oczekiwanych wyników z rzeczywistym wyjściem z sieci, zobaczysz, że wszystkie pasują całkiem blisko. I za każdym razem, gdy go uruchomisz, wyniki będą nieco inne, ponieważ na początku biegu inicjalizujesz wagi losowymi liczbami. Celem szkolenia sieci neuronowej nie jest uzyskanie dokładnego wyniku — tylko w ramach określonej tolerancji prawidłowego wyniku. Na przykład, gdybyśmy powiedzieli, że każdy wynik powyżej 0,9 to 1, a każdy wynik poniżej 0,1 to 0, wtedy nasza sieć dałaby doskonałe wyniki. Strata sumy do kwadratu jest miarą wszystkich błędów wszystkich możliwych danych wejściowych. Jeśli sporządzimy wykres sumy kwadratów strat w funkcji liczby epok, otrzymamy wykres pokazany na rysunku .



Widać, że dość szybko się poprawiamy, a potem to się kończy. Dla naszego postawionego problemu istnieje 1000 epok.

Jeszcze jeden eksperyment. Jeśli zwiększysz liczbę epok do 100 000, liczby będą jeszcze lepsze, ale nasze wyniki, zgodnie z naszymi kryteriami dokładności (> 0,9 = 1 i < 0,1 = 0), były wystarczająco dobre w przebiegu 1000 epok.

Epoch # 99999

Network Input :

[[0. 0. 0.]

[0. 0. 1.]

[0. 1. 0.]

[0. 1. 1.]

[1. 0. 0.]

[1. 0. 1.]

[1. 1. 0.]

[1. 1. 1.]]

Expected Output of XOR Gate Neural Network:

[[1.]

[0.]

[0.]

[0.]

[0.]

[0.]

[0.]

[1.]]

Actual Output from XOR Gate Neural Network:

[[9.85225608e-01]

[1.41750544e-04]

[1.51985054e-04]

[1.14829204e-02]

[1.17578404e-04]

[1.14814754e-02]

[1.14821256e-02]

[9.78014943e-01]]

Sum Squared Loss:

0.00013715041859631841

Predicted XOR output data based on trained weights:

Expected (X1-X3):

[0. 0. 1.]

Output (Y1):

[0.00014175]

Używanie TensorFlow dla tej samej sieci neuronowej

TensorFlow to pakiet Pythona, który jest również przeznaczony do obsługi sieci neuronowych opartych na macierzach i grafach przepływu podobnych do NumPy. Różni się od NumPy pod jednym zasadniczym względem: TensorFlow jest przeznaczony do użytku w aplikacjach uczenia maszynowego i sztucznej inteligencji, a więc ma biblioteki i funkcje zaprojektowane dla tych aplikacji. TensorFlow bierze swoją nazwę od sposobu, w jaki przetwarza dane. Tensor to wielowymiarowa macierz danych, która jest przekształcana przez każdą warstwę TensorFlow, przez którą przechodzi. TensorFlow jest

niezwykle przyjazny dla Pythona i może być używany na wielu różnych maszynach, a także w chmurze. Jak dowiedziałeś się z poprzedniej sekcji o sieciach neuronowych w Pythonie, sieci neuronowe są grafami przepływu danych i są implementowane w kategoriach wykonywania operacji na macierzach danych, a następnie przenoszenia wynikowych danych do innej macierzy. Ponieważ macierze są tensorami, a dane przepływają z jednej do drugiej, możesz zobaczyć, skąd wzięła się nazwa TensorFlow. TensorFlow to jedna z najlepiej obsługiwanych platform aplikacji z interfejsami API (interfejsy programowania aplikacji) gpt Python, C , Haskell, Java, Go, Rust, a także pakiet innej firmy dla R o nazwie tensorflow.

Instalowanie biblioteki Pythona TensorFlow

W przypadku systemów Windows, Linux i Raspberry Pi sprawdź oficjalny link TensorFlow pod adresem https://www.tensorflow.org/install/pip. TensorFlow to typowa biblioteka Python3 i API (interfejs programowania aplikacji). TensorFlow ma wiele zależności, które zostaną również zainstalowane, postępując zgodnie z samouczkiem, o którym mowa powyżej.

WPROWADZENIE TENSORÓW

Wiesz już, że tensory to wielowymiarowe macierze danych. Ale dodatkowe omówienie będzie pomocne w opanowaniu słownictwa tensorów. Sieci neuronowe są grafami przepływu danych i są zaimplementowanymi pośrednikami wykonywania operacji na macierzach danych, a następnie przenoszenia wynikowych danych do innej macierzy. Tensor to inna nazwa macierzy.

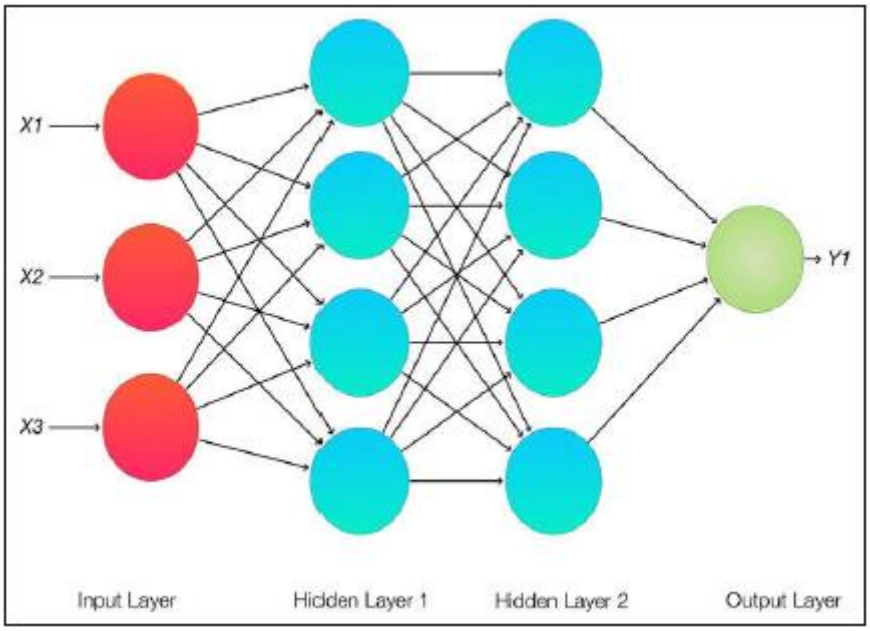
Skalary: skalar można traktować jako pojedynczą porcję danych. Ze skalarem jest powiązana jedna i tylko jedna część danych. Na przykład wartość 5 metrów, 5 metrów na sekundę to przykłady wartości skalarnych, podobnie jak 45 stopni Fahrenheita lub 21 stopni Celsjusza. Możesz myśleć o skalarze jako o punkcie na płaszczyźnie.

Wektory: Wektor różni się od skalara tym, że zawiera co najmniej dwie informacje. Przykład wektora to 5 metrów na wschód - opisuje odległość i kierunek. Wektor to jednowymiarowa macierz (na przykład 2x1). Możesz myśleć o wektorze jako o strzałce umieszczonej na płaszczyźnie. Wygląda jak promień na płaszczyźnie. Wektory płaskie są najprostszą formą tensora. Jeśli spojrzysz na wektor 3x1, teraz masz współrzędne przestrzeni 3D: x, y i z.

Tensory: Wektory to szczególne przypadki tensorów. Tensor to macierz, którą można scharakteryzować za pomocą wielkości i wielu kierunków. Skalary można rozpoznać jako pojedyncze liczby, wektory jako uporządkowane zestawy liczb, a tensory za pomocą jedno- lub wielowymiarowej tablicy liczb. Oto świetne niematematyczne wprowadzenie do tensorów: https://www.youtube.com/watch?v=f5liqUk0ZTw.

Budowanie sieci neuronowej w Pythonie w TensorFlow

W naszym przykładzie sieci neuronowej w TensorFlow użyjemy tej samej sieci, której użyliśmy do zaimplementowania bramki XOR w Pythonie. Rysunek 1 przedstawia dwuwarstwową sieć neuronową, której użyliśmy; Rysunek 5 przedstawia nową trójwarstwową sieć neuronową. Tabela 1 zawiera tabelę prawdy dla obu sieci.



TensorFlow to przyjazna dla języka Python platforma aplikacji i zbiór funkcji zaprojektowanych do zastosowań AI, zwłaszcza w sieciach neuronowych i uczeniu maszynowym. Używa Pythona, aby zapewnić przyjazny dla użytkownika, wygodny interfejs użytkownika podczas wykonywania tych aplikacji za pomocą kodu C o wysokiej wydajności. Keras to biblioteka sieci neuronowych typu open source, która umożliwia szybkie eksperymentowanie z sieciami neuronowymi, uczeniem głębokim i uczeniem maszynowym. W 2017 roku Google zdecydowało się natywnie wspierać Keras jako preferowany interfejs dla TensorFlow. Keras zapewnia doskonały i intuicyjny zestaw abstrakcji i funkcji, podczas gdy TensorFlow zapewnia wydajną podstawową implementację. Oto pięć kroków do wdrożenia sieci neuronowej w Keras z TensorFlow

1. Załaduj i sformatuj swoje dane.

2. Zdefiniuj model i warstwy sieci neuronowej.

3. Skompiluj model

4. Dopasuj i wytrenuj swojego modela.

5. Oceń model.

Ładowanie danych

Ten krok jest dość trywialny w naszym modelu, ale często jest najbardziej złożoną i najtrudniejszą częścią budowania całego programu. Musisz spojrzeć na swoje dane (czy to bramkę XOR, czy bazę danych czynników wpływających na pacjentów z cukrzycą) i dowiedzieć się, jak zmapować swoje dane i wyniki, aby uzyskać potrzebne informacje i prognozy.

Definiowanie modelu i warstw sieci neuronowej

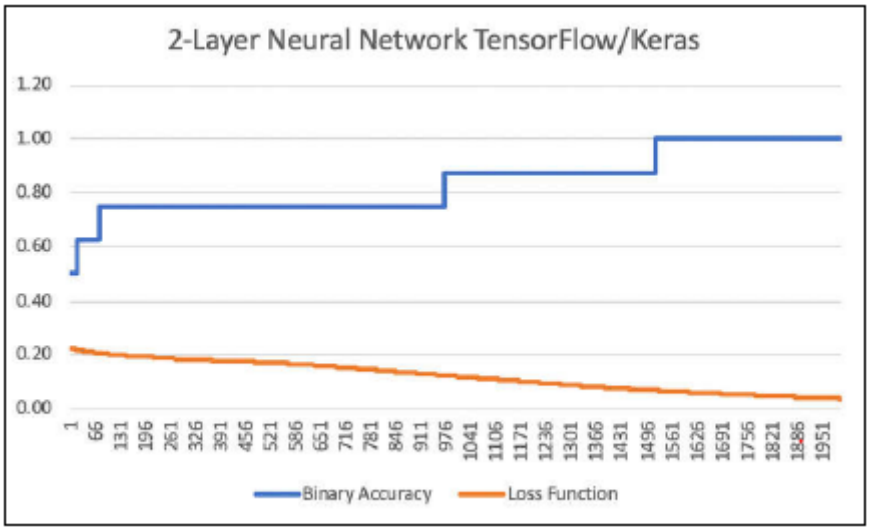
Określanie sieci jest jedną z głównych zalet Keras w porównaniu z innymi frameworkami. Zasadniczo po prostu tworzysz stos warstw neuronowych, przez które mają przepływać dane. Pamiętaj, że TensorFlow to właśnie to. Twoje macierze danych przepływających przez stos sieci neuronowej. Tutaj wybrałeś konfigurację swojej warstwy neuronowej i funkcji aktywacji.

Kompilowanie modelu

Następnie kompilujesz swój model, który łączy twój model warstwy Keras z wydajną bazą (nazywaną zapleczem) w celu uruchomienia na twoim sprzęcie. Ty także wybierasz, czego chcesz użyć dla funkcji straty.

Dopasowanie i szkolenie Twojego modelu

To tutaj odbywa się prawdziwa praca polegająca na szkoleniu sieci. Sam określisz, przez ile epok chcesz przejść. Gromadzi również historię tego, co dzieje się we wszystkich epokach, a my wykorzystamy to do stworzenia naszych wykresów. Poniżej znajduje się nasz kod Pythona wykorzystujący TensorFlow, NumPy i Keras dla dwuwarstwowej sieci neuronowej z rysunku.



Używając nano (lub swojego ulubionego edytora tekstu), otwórz plik o nazwie TensorFlowKeras.py i wprowadź następujący kod:import tensorflow as tf

from tensorflow.keras import layers

from tensorflow.keras.layers import Activation, Dense

import numpy as np

X = input of our 3 input XOR gate

set up the inputs of the neural network (right from the table)

X = np.array(([0,0,0],[0,0,1],[0,1,0],

[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)

y = our output of our neural network

y = np.array(([1], [0], [0], [0], [0],

[0], [0], [1]), dtype=float)

model = tf.keras.Sequential()

model.add(Dense(4, input_dim=3, activation='relu',

```
use_bias=True))
#model.add(Dense(4, activation='relu', use_bias=True))
model.add(Dense(1, activation='sigmoid', use_bias=True))
model.compile(loss='mean_squared_error',
optimizer='adam',
metrics=['binary_accuracy'])
print (model.get_weights())
history = model.fit(X, y, epochs=2000,
validation_data = (X, y))
model.summary()
# printing out to file
loss_history = history.history["loss"]
numpy_loss_history = np.array(loss_history)
np.savetxt("loss_history.txt", numpy_loss_history,
delimiter="\n")
binary_accuracy_history = history.history["binary_accuracy"]
numpy_binary_accuracy = np.array(binary_accuracy_history)
np.savetxt("binary_accuracy.txt", numpy_binary_accuracy, delimiter="\n")
print(np.mean(history.history["binary_accuracy"]))
result = model.predict(X ).round()
print (result)
```

Po zapoznaniu się z kodem uruchomimy sieć neuronową, a następnie ocenimy model i wyniki.

Łamanie kodu

Pierwszą rzeczą, na którą należy zwrócić uwagę w naszym kodzie, jest to, że jest on znacznie prostszy niż nasz dwuwarstwowy model ściśle w Pythonie, użyty wcześniej w tej części. Na tym polega magia TensorFlow/Keras. Wróć i porównaj ten kod z kodem naszej dwuwarstwowej sieci w czystym Pythonie. Jest to o wiele prostsze i łatwiejsze do zrozumienia. Najpierw importujemy wszystkie biblioteki potrzebne do uruchomienia naszego przykładowego modelu dwuwarstwowego. Zauważ, że TensorFlow domyślnie zawiera Keras. I po raz kolejny widzimy naszego przyjaciela NumPy jako preferowany sposób obsługi macierzy.

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.layers import Activation, Dense
```

import numpy as np.

Krok 1, załaduj i sformatuj swoje dane. W tym przypadku właśnie ustawiliśmy tablicę prawdy dla naszej bramki XOR pod względem tablic NumPy. Może to stać się znacznie bardziej złożone, gdy masz duże, zróżnicowane, skorelowane krzyżowo źródła danych.

```
# X = input of our 3 input XOR gate

# set up the inputs of the neural network (right from the table)

X = np.array(([0,0,0],[0,0,1],[0,1,0],

[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)

# y = our output of our neural network

y = np.array(([1], [0], [0], [0], [0],

[0], [0], [1]), dtype=float)
```

Krok 2, zdefiniuj model i warstwy sieci neuronowej. W tym tkwi prawdziwa moc Keras. Dodawanie kolejnych warstw neuronowych oraz zmiana ich rozmiaru i funkcji aktywacji jest bardzo prosta. Stosujemy również obciążenie do naszej funkcji aktywacji (w tym przypadku relu z naszym przyjacielem sigmoidą dla końcowej warstwy wyjściowej), czego nie zrobiliśmy w naszym czystym modelu Pythona. Zobacz skomentowaną instrukcję model.add poniżej? Kiedy przejdziemy do naszego przykładu trójwarstwowej sieci neuronowej, to wszystko, co musimy zmienić, usuwając komentarz.

```
model = tf.keras.Sequential()

model.add(Dense(4, input_dim=3, activation='relu',

use_bias=True))

#model.add(Dense(4, activation='relu', use_bias=True))

model.add(Dense(1, activation='sigmoid', use_bias=True))
```

Krok 3, skompiluj swój model. Używamy tej samej funkcji straty, której użyliśmy w naszej implementacji w czystym Pythonie, mean_squared_error. Nowością jest dla nas optymalizator, ADAM (metoda optymalizacji stochastycznej) jest dobrym domyślnym optymalizatorem. Zapewnia metodę skutecznego zmniejszania gradientu zastosowanego do ciężarów warstw. Należy zwrócić uwagę na to, o co prosimy w zakresie wskaźników. binary_accuracy oznacza, że porównujemy nasze wyniki z naszej sieci z 1 lub 0. Zobaczysz wartości, powiedzmy, 0,75, co, ponieważ mamy osiem możliwych wyników, oznacza, że sześć z ośmiu jest poprawnych. Jest dokładnie tym, czego można się spodziewać po nazwie.

```
model.compile(loss='mean_squared_error',

optimizer='adam',

metrics=['binary_accuracy'])
```

Tutaj drukujemy wszystkie masy startowe naszego modelu. Zwróć uwagę, że są one przypisane z domyślną losową metodą, którą możesz wysiać (aby wykonać ten sam bieg z tymi samymi ciężarami startowymi za każdym razem) lub możesz zmienić sposób ich dodawania

```
print (model.get_weights())
```

Krok 4, dopasuj i wytrenuj swojego modela. Wybraliśmy liczbę epok, abyśmy przez większość czasu uzyskali zbieżność z binarną dokładnością 1,0. Tutaj ładujemy tablice NumPy dla danych wejściowych do naszej sieci (X) i oczekiwanego wyjścia sieci (y). Parametr validation_data służy do porównywania danych wyjściowych wytrenowanej sieci w każdej epoce i generuje val_acc i val_loss dla twoich informacji w każdej epoce, zgodnie z zapisem w zmiennej historii.

```
history = model.fit(X, y, epochs=2000,

validation_data = (X, y))
```

Tutaj wydrukujemy podsumowanie Twojego modelu, abyś mógł upewnić się, że został skonstruowany w oczekiwany sposób.

```
model.summary()
```

Następnie drukujemy wartości ze zmiennej historii, które chcielibyśmy wykreślić.

```
# printing out to file

loss_history = history.history["loss"]

numpy_loss_history = np.array(loss_history)

np.savetxt("loss_history.txt", numpy_loss_history,

delimiter="\n")

binary_accuracy_history = history.history["binary_accuracy"]

numpy_binary_accuracy = np.array(binary_accuracy_history)

np.savetxt("binary_accuracy.txt", numpy_binary_accuracy, delimiter="\n")
```

Krok 5, oceń model. Tutaj uruchamiamy model, aby przewidzieć wyjścia ze wszystkich wejść X, używając funkcji round, aby ustawić je na 0 lub 1. Zauważ, że to zastępuje kryteria, których użyliśmy w naszym czystym modelu Pythona, który wynosił <0,1 = „0” i >0,9 = „1”. Obliczyliśmy również średnią wszystkich wartości binary_accuracy wszystkich epok, ale liczba ta nie jest zbyt użyteczna — poza tym, że im jest bliższa 1,0, tym szybciej model się powiódł.

```
print(np.mean(history.history["binary_accuracy"]))

result = model.predict(X ).round()

print (result)
```

Przejdźmy teraz do niektórych wyników.

Ocena modelu

Podczas uruchamiania programów TensorFlow możesz zobaczyć coś takiego:

```
usr/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning: compiletime

version 3.4 of module 'tensorflow.python.framework.fast_tensor_util' does not

match runtime version 3.5

return f(*args, **kwds)
```

/usr/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning: builtins.type

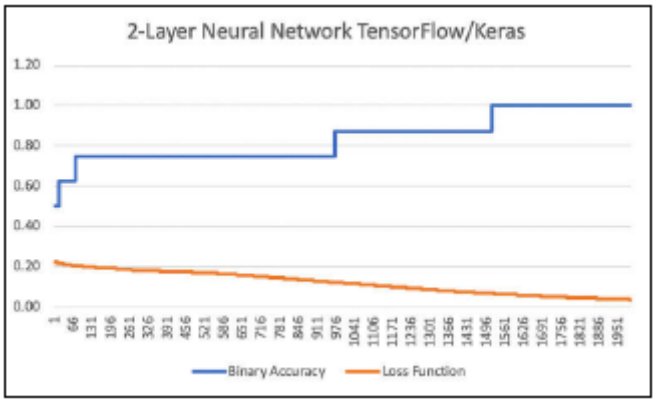
size changed, may indicate binary incompatibility. Expected 432, got 412

return f(*args, **kwds)

Wynika to z problemu ze sposobem, w jaki TensorFlow został zbudowany dla twojej maszyny. Te ostrzeżenia można bezpiecznie zignorować. Dobrzy ludzie z TensorFlow.org twierdzą, że ten problem zostanie rozwiązany w następnej wersji. Uruchamiamy model dwuwarstwowy, wpisując python3 TensorFlowKeras.py w naszym oknie terminala. Po obejrzeniu odchodzących epok (możesz zmienić tę ilość danych wyjściowych, ustawiając parametr Verbose w swoim poleceniu model.fit), zostaniemy nagrodzeni następującymi nagrodami:

```
...
Epoch 1999/2000
8/8 [==============================] - 0s 2ms/step - loss: 0.0367 - binary_
   accuracy: 1.0000 - val_loss: 0.0367 - val_binary_accuracy: 1.0000
Epoch 2000/2000
8/8 [==============================] - 0s 2ms/step - loss: 0.0367 - binary_
   accuracy: 1.0000 - val_loss: 0.0367 - val_binary_accuracy: 1.0000
_________________________________________________________________
Layer (type)        Output Shape       Param #
=================================================================
dense (Dense)        (None, 4)        16
_________________________________________________________________
dense_1 (Dense)       (None, 1)        5
=================================================================
Total params: 21
Trainable params: 21
Non-trainable params: 0
_________________________________________________________________
0.8436875
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]]
```

Widzimy, że do epoki 2000 osiągnęliśmy dokładność binarną 1,0, zgodnie z oczekiwaniami, a wyniki naszego wywołania funkcji model.predict na końcu pasują do naszej tabeli prawdy. Rysunek przedstawia wyniki funkcji utraty i wartości dokładności binarnej wykreślone w funkcji numeru epoki w miarę postępu uczenia.



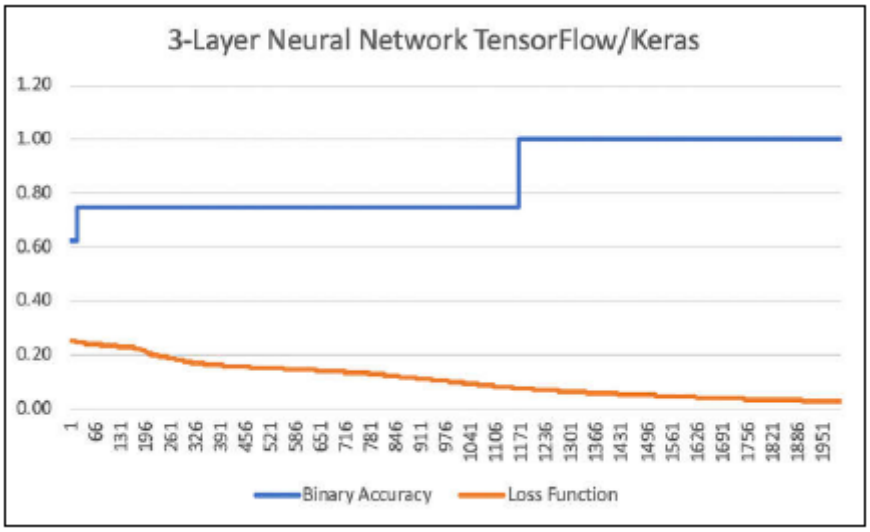
Rysunek 2 pokazuje graficznie, co wdrażamy. Kilka rzeczy do zapamiętania. Funkcja straty jest znacznie gładszą krzywą liniową, gdy się powiedzie. Ma to związek z wyborem aktywacji (relu) i funkcją optymalizatora (ADAM). Kolejną rzeczą do zapamiętania jest to, że za każdym razem otrzymasz inną krzywą (trochę) ze względu na losowe liczby początkowe wartości wag. Zasiej swój generator liczb losowych, aby był taki sam za każdym razem, gdy go uruchomisz. Ułatwia to optymalizację wydajności.

PROPAGACJA WSTECZNA W KERAS

Z naszą pierwszą siecią neuronową w tej części zrobiliśmy wiele o wstecznej propagacji i o tym, jak była to podstawowa część sieci neuronowych. Jednak teraz przenieśliśmy się do Keras/TensorFlow i nie powiedzieliśmy o tym ani słowa. Powodem tego jest to, że wsteczna propagacja w Keras/TensorFlow jest obsługiwany automatycznie. To dla ciebie zrobione. Jeśli chcesz zmodyfikować sposób, w jaki to robi, najprostszym sposobem jest modyfikacja parametru optymalizacji w module , polecenie kompilacji (użyliśmy programu ADAM). Radykalna modyfikacja algorytmu wstecznej propagacji w Keras wymaga sporo pracy, ale jest możliwa. Kiedy przeprowadzasz szkolenie dla sieci, używasz algorytmu propagacji wstecznej i optymalizujesz go zgodnie z wybranym algorytmem optymalizacji i funkcją strat określoną podczas kompilacji modelu. Zwróć uwagę, kiedy dokładność binarna dochodzi do 1,00 (około epoki 1556). Wtedy Twoja sieć jest w pełni przeszkolona w tym przypadku.

Zmiana na trójwarstwową sieć neuronową w TensorFlow/Keras

Teraz dodajmy kolejną warstwę do naszej sieci neuronowej, jak pokazano na rysunku.



Otwórz swój plik TensorFlowKeras.py w swoim ulubionym edytorze i zmień następujące elementy:

model.add(Dense(4, input_dim=3, activation='relu',

use_bias=True))

#model.add(Dense(4, activation='relu', use_bias=True))

model.add(Dense(1, activation='sigmoid', use_bias=True))

Usuń znak komentarza przed środkową warstwą, a otrzymasz trójwarstwową sieć neuronową z czterema neuronami na warstwę. To jest takie proste. Oto jak powinien wyglądać teraz:

model.add(Dense(4, input_dim=3, activation='relu',

use_bias=True))

model.add(Dense(4, activation='relu', use_bias=True))

model.add(Dense(1, activation='sigmoid', use_bias=True))

Uruchom program, a otrzymasz teraz wyniki z trójwarstwowej sieci neuronowej, które będą wyglądać mniej więcej tak:

```
8/8 [==============================] - 0s 2ms/step - loss: 0.0153 - binary_
   accuracy: 1.0000 - val_loss: 0.0153 - val_binary_accuracy: 1.0000
Epoch 2000/2000
8/8 [==============================] - 0s 2ms/step - loss: 0.0153 - binary_
   accuracy: 1.0000 - val_loss: 0.0153 - val_binary_accuracy: 1.0000
_________________________________________________________________
Layer (type)        Output Shape       Param #
=================================================================
dense (Dense)        (None, 4)         16
_________________________________________________________________
dense_1 (Dense)      (None, 4)         20
_________________________________________________________________
dense_2 (Dense)      (None, 1)         5
=================================================================
Total params: 41
Trainable params: 41
Non-trainable params: 0
_________________________________________________________________
0.980375
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]]
```

DLACZEGO WARTO UŻYWAĆ GUI (GRAFICZNEGO INTERFEJSU UŻYTKOWNIKA) DO URUCHOMIENIA TensorFlow?

Jak już powinieneś wiedzieć, spędzasz dużo czasu na kodowaniu w edytorach tekstu, aby zbudować swoje modele. Dla uproszczenia, wyeksportowaliśmy nasze dane do programu Excel, aby utworzyć wykresy w tej części. Przez większość czasu używamy naszych maszyn w oknie terminala, ale używanie pełnego pulpitu GUI komputera do otwierania okien seterminal do edycji ma dużą zaletę. Ta duża zaleta nazywa się TensorBoard. TensorBoard jest częścią TensorFlow i jest dostępny w przeglądarce

takiej jak Chrome czy Firefox. Wskazujesz TensorBoard na swój katalog zadań i nagle możesz przeprowadzić wszelkiego rodzaju analizę wizualną swoich eksperymentów z siecią neuronową.

Teraz możesz zobaczyć, że masz trzy warstwy w swojej sieci neuronowej. To jeden z powodów, dla których oprogramowanie TensorFlow/Keras jest tak potężne. Łatwo jest majstrować przy parametrach i wprowadzać zmiany. Uwagi na temat naszego przebiegu trójwarstwowego: Po pierwsze, zbiega się on z binarną dokładnością 1,00 około epoki 916, znacznie szybciej niż epoka 1556 z naszego przebiegu dwuwarstwowego. Funkcja straty jest mniej liniowa niż przebieg dwuwarstwowy. Dla zabawy i śmiechu zmieniliśmy liczbę neuronów na 100 na każdą z ukrytych warstw. Zgodnie z oczekiwaniami, w epoce 78 zbiegła się z binarną dokładnością 1,00, znacznie szybciej niż wcześniejszy przebieg! Przeprowadź własne eksperymenty, aby uzyskać dobre wyczucie sposob, w jaki wyniki będą się różnić w zależności od różnych parametrów, warstw i liczby neuronów. Wierz lub nie, ale teraz już dużo wiesz o tym, jak działają sieci neuronowe i uczenie maszynowe. Idź i trenuj te neurony!