Andrew Ng Course Note 6

Shu Wang

May 31 2023

1 Advanced Learning Algorithms

Inference(prediction), training, practical advice for building machine learning systems, Decision Trees

Neural Networks:

Origin: algorithms that try to mimic the brain.

Used in the 1980's and early 1990's.

Resurgence from around 2005.

First application: speech recognition.

speech $\longrightarrow$ images $\longrightarrow$ text(NLP)

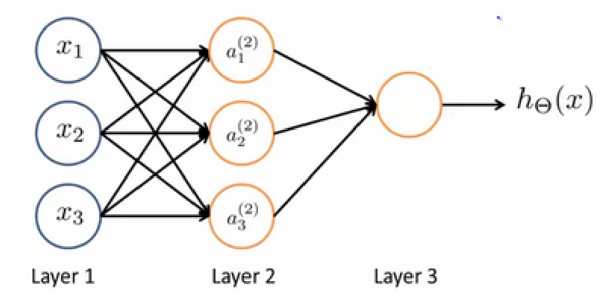
Performance keeps going up for big data.

2 Demand Prediction

x = price input

Output: $\frac{1}{1+e^{-(wx+b)}}$

Activation function.



Input Layer $\longrightarrow$ Hidden Layer $\longrightarrow$ Output Layer

A 1000 x 1000 pixels graph stands for a 1000 rows x 1000 column matrix.

Unroll it into a vector, you end up with a vector of 1 million numbers.

The first hidden unit:

$\overrightarrow{w_1}, b_1$

$a_1 = g(\overrightarrow{w_1} \cdot \overrightarrow{x} + b_1)$

where $g(z) = \frac{1}{1+e^{-z}}$

The second hidden unit:

$\overrightarrow{w_2}, b_2$

$a_2 = g(\overrightarrow{w_2} \cdot \overrightarrow{x} + b_2)$

where $g(z) = \frac{1}{1+e^{-z}}$

We use $\overrightarrow{a}^{[1]}$ to denote the output of the first hidden layer.

$a_1^{[2]} = g(\overrightarrow{w_1} \cdot \overrightarrow{a}^{[1]} + b_1)$

We use $\overrightarrow{a}^{[2]}$ to denote the output of the second hidden layer.

If it is a binary prediction, set a threshold.

Each layer/neuron has a unique function.

$a_j^{[l]} = g(\overrightarrow{w_j}^{[l]} \cdot \overrightarrow{a}^{[l-1]} + b_j^{[l]})$

g: activation function

3 Tensorflow

Create an array:

```
x = np.array([[200.0,17.0]])


layer_1 = Dense(Units = 3, activation = 'sigmoid')
a1 = layer_1(x)


layer_2 = Dense(Units = 1, activation = 'sigmoid')
a2 = layer_2(a1)
```

Note about numpy arrays:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

In numpy:

```
x = np.array([[1,2,3],[4,5,6]])
```

$$\begin{pmatrix} 0.1 & 0.2 \\ -3 & -4 \\ -0.5 & -0.6 \\ 7 & 8 \end{pmatrix}$$

In numpy:

```
x = np.array([[0.1,0.2],[-3.0, -4.0,],[-0.5, -0.6,],[7.0,8.0,]])
```

```
x = np.array([[200,17]])
```

creates a row vector:

$$\begin{pmatrix} 200 & 17 \end{pmatrix}$$

```
x = np.array([[200],[17]])
```

creates a column vector:

$$\begin{pmatrix} 200 \\ 17 \end{pmatrix}$$

Tensor $\longrightarrow$ numpy

```
layer_1 = Dense(Units = 3, activation = 'sigmoid')
layer_2 = Dense(Units = 1, activation = 'sigmoid')
model = Sequential([layer_1,layer_2])
```

Call two functions:

```
x = np.array([[200,17],[120,5],[425,20],[212,18]])
```

```
y = np.array([1,0,0,1])
model.compile(...)
model.fit(x,y)
model.predict(x_new)
```

```
x = np.array([200,17])
```

$a_1^{[1]} = g(\overrightarrow{w_1}^{[1]} \cdot \overrightarrow{x} + b_1^{[1]})$

```
w1_1 = np.array([1,2])
b1_1 = np.array([-1]
z1_1 = np.dot(w1_1,x)+b
a1_1 = sigmoid(z1_1)
```

$a_2^{[1]} = g(\overrightarrow{w_2}^{[1]} \cdot \overrightarrow{x} + b_2^{[1]})$

```
w1_2 = np.array([-3,4])
b1_2 = np.array([1]
z1_2 = np.dot(w1_2,x)+b
a1_2 = sigmoid(z1_2)
```

$a_3^{[1]} = g(\overrightarrow{w_3}^{[1]} \cdot \overrightarrow{x} + b_3^{[1]})$

```
w1_3 = np.array([5, -6])
b1_3 = np.array([2]
z1_3 = np.dot(w1_3, x)+b
a1_3 = sigmoid(z1_3)
```

$a_1^{[2]} = g(\overrightarrow{w_1}^{[2]} \cdot \overrightarrow{a^{[1]}} + b_1^{[2]})$

```
w2_1 = np.array([-7, 8])
b2_1 = np.array([3]
z2_1 = np.dot(w2_1, a1)+b2_1

def dense(a_in, W, b, g):
W = np.array([[1, -3, 5], [2, -4, 6]])    2 by 3
b = np.array([-1, 1, 2])


a_in = np.array([-2, 4])


  Units = W.shape[1]    #Units = 3
  a_out = np.zeros(units) # a = [0,0,0]
  for j in range(units): # j = 0,1,2
    w = W[:, j]          #Take the j-th column of a matrix
    z = np.dot(w, a_in) + b[j]
    a_out[j] = g(z)
  return a_out


def sequential(x):
  a1 = dense(x, W1, b1)
  a2 = dense(a1, W2, b2)
  a3 = dense(a2, W3, b3)
  a4 = dense(a3, W4, b4)
  f_x = a4
  return f_x
```

artificial narrow intelligence, (E.g smart speaker, self-driving car,web search, AI in farming and factories)

artificial general intelligence, (E.g Do anything a human can do)

Neural networks can be implemented efficiently using matrix multiplication.

4 Vectorized Neural Network

```
def dense(a_in,W,b,g):
W = np.array([[1, -3,5],[2, -4,6]])    2 by 3
b = np.array([-1,1,2])
X = np.array([[200,17]])

def dense(a_in,W,b):

  Units = W.shape[1]   #Units = 3
  a_out = np.zeros(units) # a = [0,0,0]
  for j in range(units): # j = 0,1,2
    w = W[:,j]           #Take the j-th column of a matrix
    z = np.dot(w,a_in) + b[j]
    a_out[j] = g(z)
  return a_out
```

Another way in matrix:

```
def dense(A_in,W,b,g):
W = np.array([[1, -3,5],[2, -4,6]])    2 by 3
b = np.array([-1,1,2])
X = np.array([[200,17]])



  Z = np.matmul(A_in,W) +B
  A_out = g(Z)
  return A_out
```

Dot products:

example:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

$z = (1 * 3) + (2 * 4) = 11$

In general, $z = \overrightarrow{a} \cdot \overrightarrow{w}$

Transpose: $\overrightarrow{a} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$

$\overrightarrow{a}^T = \begin{bmatrix} 1 & 2 \end{bmatrix}$

Vector multiplication:

$z = \overrightarrow{a}^T \overrightarrow{w}$

Vector matrix multiplication:

$\overrightarrow{a} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$

$\overrightarrow{a}^T = \begin{bmatrix} 1 & 2 \end{bmatrix}$

$W = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$

Z = $[\overrightarrow{a}^T \overrightarrow{w_1}, \overrightarrow{a}^T \overrightarrow{w_2}]$

$A = \begin{bmatrix} 1 & -1 \\ 2 & -2 \end{bmatrix}$, $A^T = \begin{bmatrix} 1 & 2 \\ -1 & -2 \end{bmatrix}$

$W = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$

$Z = A^T W = \begin{bmatrix} \overrightarrow{a_1}^T \overrightarrow{w_1} & \overrightarrow{a_1}^T \overrightarrow{w_2} \\ \overrightarrow{a_2}^T \overrightarrow{w_1} & \overrightarrow{a_2}^T \overrightarrow{w_2} \end{bmatrix}$

Matrix Multiplication rules

$A = \begin{bmatrix} 1 & -1 & 0.1 \\ 2 & -2 & 0.2 \end{bmatrix}$, $A^T = \begin{bmatrix} 1 & 2 \\ -1 & -2 \\ 0.1 & 0.2 \end{bmatrix}$

$W = \begin{bmatrix} 3 & 5 & 7 & 9 \\ 4 & 6 & 8 & 0 \end{bmatrix}$,

$Z = A^T W = \begin{bmatrix} 11 & 17 & 23 & 9 \\ -11 & -17 & -23 & -9 \\ 1.1 & 1.7 & 2.3 & 0.9 \end{bmatrix}$

Z is a 3 by 4 matrix; same number of rows as A^T and same number of columns as W

```
A = np.array([1,-1,0.1],[2,-2,0.2]])

AT = np.array([1,2],[-1,-2],[0.1,0.2])

AT = A.T

W = np.array([3,5,7,9],[2,4,8,0])

Z = np.matmul(AT,W)

# or you can write Z = AT @ W
```

$A^T = \begin{bmatrix} 200 & 17 \end{bmatrix}$ $W = \begin{bmatrix} 1 & -3 & 5 \\ 2 & -4 & 6 \end{bmatrix}$ $\overrightarrow{b} = \begin{bmatrix} -1 & 1 & 2 \end{bmatrix}$

In code:

```
AT = np.array([200,17])

W = np.array([1,-3,5],[-2,4,6])
b = np.array([[-1,1,2]])


def dense(AT,Wb,g)

  z = np.matmul(AT,W) + b

  a_out = g(z)

  return a_out
```