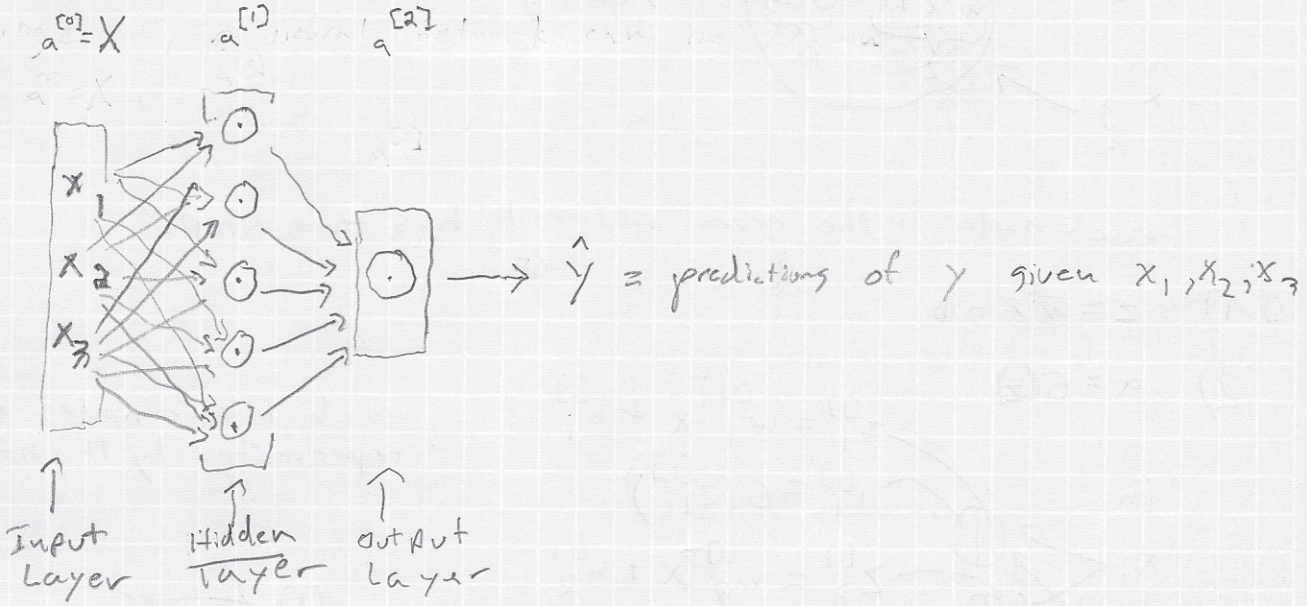
Deep Learning.ai Coursera Specialization in Deep Learning Course 1

Neural Networks and Deep Learning - Dr. Andrew Ng

Week 3 - Shallow Neural Networks

Neural Network Representations



The hidden layer means you don't see them in the training set.

In this course what we were calling X now we represent X as $a^{[0]}$

$$X = a^{[0]}$$

and " $\hat{y} = a$ "

This is called a 2 layer neural network because we don't count the Input layer (by convention).

The hidden layer will also have associated with $w^{[1]}, b^{[1]}$

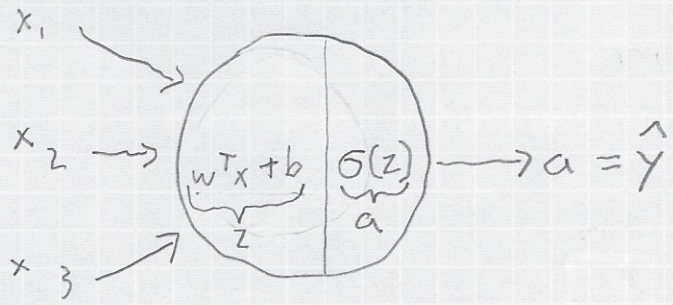
In this example $w^{[1]}$ will be a $(4,3)$ matrix — 3 input features, 4 nodes / hidden units

$b^{[1]}$ will be a $(4,1)$ vector

The outpts $w^{[2]}$ will be a $(1,4)$ — 4 input hidden units, 1 output

and $b^{[2]}$ " " " $(1,1)$

- Computing a Neural Network's Output



each node in the neural network has two steps

1) $z = w^T x + b$

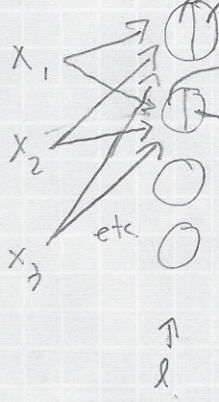
2) $a = \sigma(z)$

$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$

$a_1^{[1]} = \sigma(z_1^{[1]})$

$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$

$a_2^{[1]} = \sigma(z_2^{[1]})$



each layer's nodes are represented by this notation

$a_i^{[l]}$ ← layer

← node in layer

In this example, with 4 nodes we want to compute all z's and a's at the same time, so we transpone our column vectors into rows, and stack them as follows:

$$\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$\langle 4, 3 \rangle$ ← matrix multiplication

We have 4 logistic regression units, and each unit has a corresponding vector w by stacking them you get a 4 by 3 Matrix

$W^{[1]}$

$Z^{[1]}$

We stack nodes vertically

So, given input x:

$a^{[0]}$ $\quad \hat{y} = a^{[2]}$

first layer

$$z^{[1]} = W^{[1]} x + b$$
$$(4,1) \quad (4,3)\,(3,1)\,(4,1)$$

$$a^{[1]} = \sigma(z^{[1]})$$
$$(4,1) \quad (4,1)$$

second layer

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$(1,1) \quad (1,4)\,(4,1)\,(1,1)$$

$$a^{[2]} = \sigma(z^{[2]})$$
$$(1,1) \quad (1,1)$$

to compute the output of a 2 layer nn for a single training example.

Vectorizing Across Multiple Examples

$x \longrightarrow a^{[2]} = \hat{y}$ for a single training example

$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$

$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$

$\vdots$

$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$

$a^{[2](i)}$ ← example i, layer 2

For $i = 1$ to m:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

} train on all examples m

but this is an inefficient for loop, let's vectorize it

we stack the x, z, and a vectors into matricies

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$
$$(n_x, m)$$

$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

Vertical index corresponds to different nodes in a nn

horizontally we index across training examples, when you sweep left to right you are scanning through the training set

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$A^{[1]} = \begin{bmatrix} \odot & \\ \odot & \end{bmatrix}$

The value at the topmost corner of the matrix corresponds to the activation of the first hidden unit on the first training example

one value down corresponds to the second hidden unit on the first traing example

↓ scanning down indexs into the hidden unit's number

→ training examples(m)

holds true for Z and also X

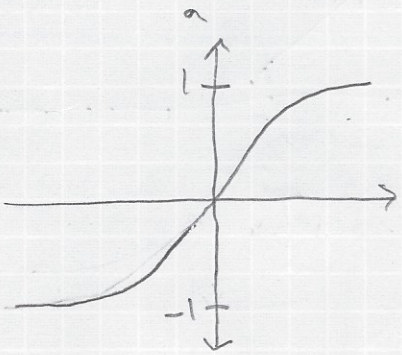
Activation Functions

You choose which functions to use for each layer and the output.

The hyperbolic tangent function almost always works better than the sigmoid function.

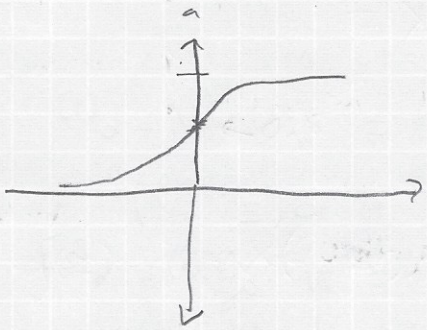
$$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

tanh centers your data around zero which makes learning for the next layer easier



vs

σ
sigmoid

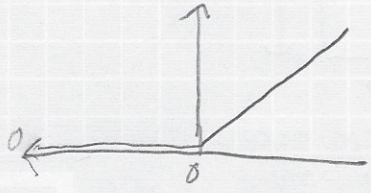


The one exemption would be for the output in Binary Classification.

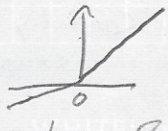
we use $g(z)$ to signify the activation

If z is very large or very small then the gradient/derivative/slope will be very small so it goes very slowly.
In these cases we can use ReLU = rectified linear Unit

$a = \max(0, z)$ → very common now in hidden units



ReLU



Leaky ReLU

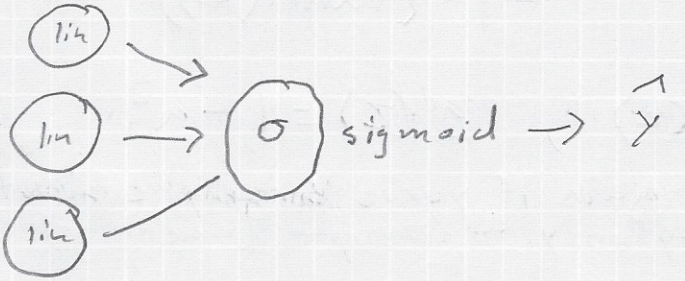
Why do you need non-linear activation functions?

If you use a linear function (or no function) for $g(z) = z$ ← the "linear activation function" (could call this "identity activation function")

If you do this it turns out that this model is just computing $\hat{y}$ as a linear function of your input features x.

If you were to use linear activation functions, or alternatively have no activation function then the NN is outputting a linear function of the input. No matter how many layers, all its doing is computing a linear activation function, so you might as well not have any hidden layers.

Even if you have a model:



it is no more expresive than standard logistic regression without any hidden layers. So linear regression hidden unit layer is useless. The composition of 2 linear functions is itself a linear function.

In the single case of doing ML on a regression problem where $y \in \mathbb{R}$ (y is a real number)

e.g. Housing price prediction where y $0 ..... $1,000,000s

If y takes on real values then it might be okay to have a linear output activation function. But then the hidden units should not be linear functions (could be ReLU, or tanh, or leaky ReLU, etc.),

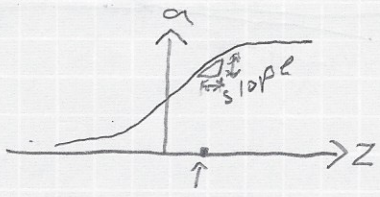
But even in this case you could use a ReLU because housing prices are never negative.

Derivatives of Activation Functions

You need to know this for understanding gradient descient.

Sigmoid activation function derivative

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



$$\frac{d}{dz} g(z) = \text{slope of } g(x) \text{ at } z$$

look up or research calculus proof

$$= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$

$$= g(z)\left(1 - g(z)\right)$$

also call $g'(z) = a(1-a)$

Tanh activation function derivative

$$g(z) = \tanh(z)$$

$$= \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

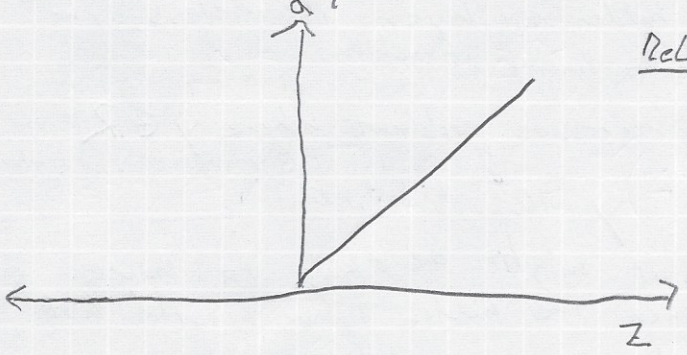
$$g'(z) = \frac{d}{dz} g(z) = \text{slope of } g(z) \text{ at } z$$

$$= 1 - (\tanh(z))^2$$

if $a = g(z)$, $g'(z) = 1 - a^2$

So once again if you've already computed the value of a this is very easy to compute (same thing we saw with the sigmoid function)

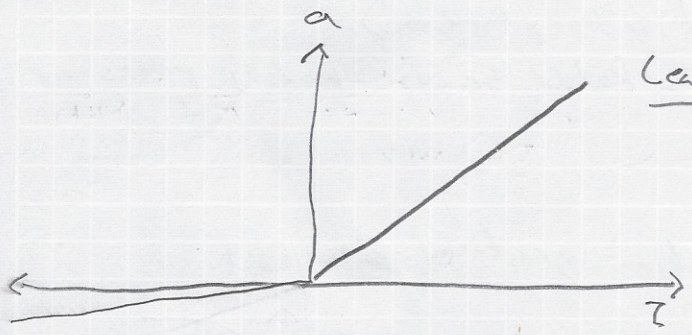
ReLU and Leaky ReLU derivatives



ReLU → $g(z) = \max(0, z)$

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$$

undefined if $z = 0$



Leaky ReLU

$g(z) = \max(0.01z, z)$

$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Gradient descent for neural networks

We're just going to give you the equations you need to implement in order to get gradient descent, our back propogation working. In the next section we'll give some intuitions for why these particular equations are the accurate equations, the correct equations, for computing the gradients you need for your NN.

NN with a single hidden layer.

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$

if you have $n_x = n^{[0]}$ (input features), $n^{[1]}$ (hidden units), $n^{[2]} = 1$ (output units)

then

$W^{[1]}$ is matrix $(n^{[1]}, n^{[0]})$

$b^{[1]}$ $(n^{[1]}, 1)$ ← an $n^{[1]}$-dimensional vector

$W^{[2]}$ $(n^{[2]}, n^{[1]})$

$b^{[2]}$ $(n^{[2]}, 1)$

Cost function (assuming we're doing binary classification)

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}, y)$$

($\hat{y} = a^{[2]}$)

Gradient descent

This optimizes the parameters with respect to the cost function. In gradient descent it's important to randomize the parameter initialization, rather than to all 0s as we did earlier in this course.

Repeat

Compute predictions $(\hat{y}^{(i)}, i = 1 \ldots m)$

Compute derivatives $dW^{[1]} = \frac{dJ}{dW^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, ...

g.d. update

$$W^{[1]} = W^{[1]} - \alpha\, dW^{[1]}$$

$$b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$$

(α ← learning rate)

$W^{[2]} \ldots$

$b^{[2]} \ldots$

you do this until the parameters looks like they are converging. The key you need is how to calculate the partial derivatives.

Forward Propogation "left to right"

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$ ← conducted element-wise
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

vectorized across your training set (that's why capital ZWXA)

activation function is sigmoid when doing binary classification

Back Propation (derivatives) "right to left"

$$dZ^{[2]} = A^{[2]} - Y$$

$Y = [y^{(1)}, y^{(2)} \cdots y^{(m)}]$ again this is vectorized across all examples.

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis=1, keepdims=True)$$

prevents output of $(n,)$ rank one array

$(n^{[2]}, 1)$

$$dz^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$$

element-wise product

$(n^{(1)}, m)$ $(n^{(1)}, m)$

$$dw^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis=1, keepdims=True)$$

$(n^{[1]}, 1)$ vector