# Coursera Deep Learning Course 1 Week 3 notes: Shallow neural networks

2017-10-10

notes deep learning

## Shallow Neural Network

### Neural Networks Overview

$^{[i]}$: layer.

$^{(i)}$: training example.

### Neural Networks Representation

$a^{[0]}$ = X: activation units of input layer.

When we count layers in neural networks, **we don't count the input layer.**

Just a recap from Machine Learning course: the hidden layers i and the output layer i will have parameters $W^{[i]}$, $b^{[i]}$ associated with them. (Unlike the past convention, the index is increased by 1).

### Computing a Neural Network's Output

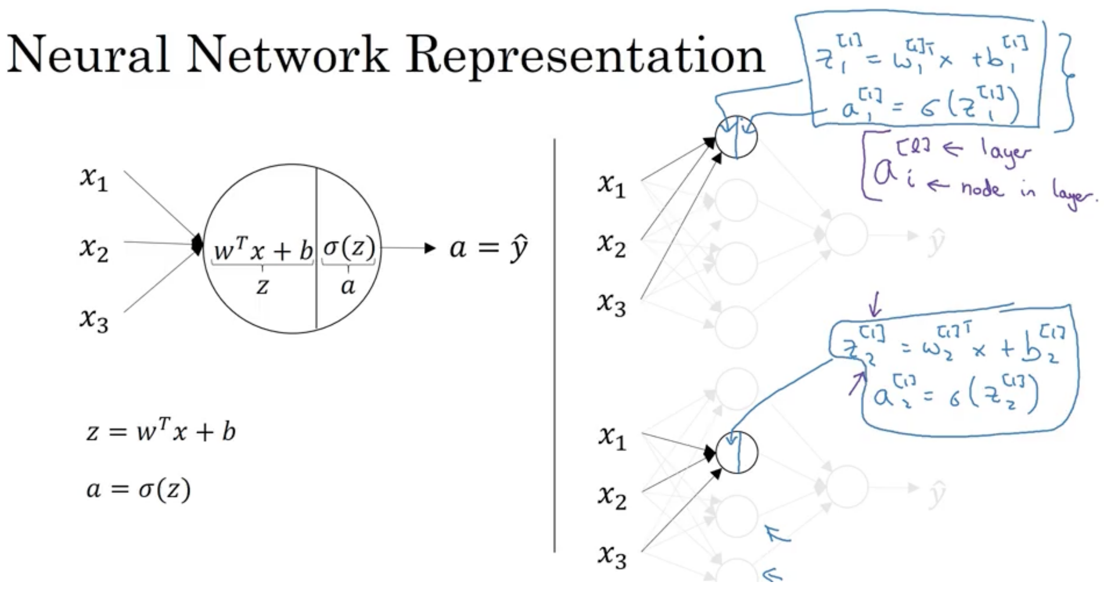
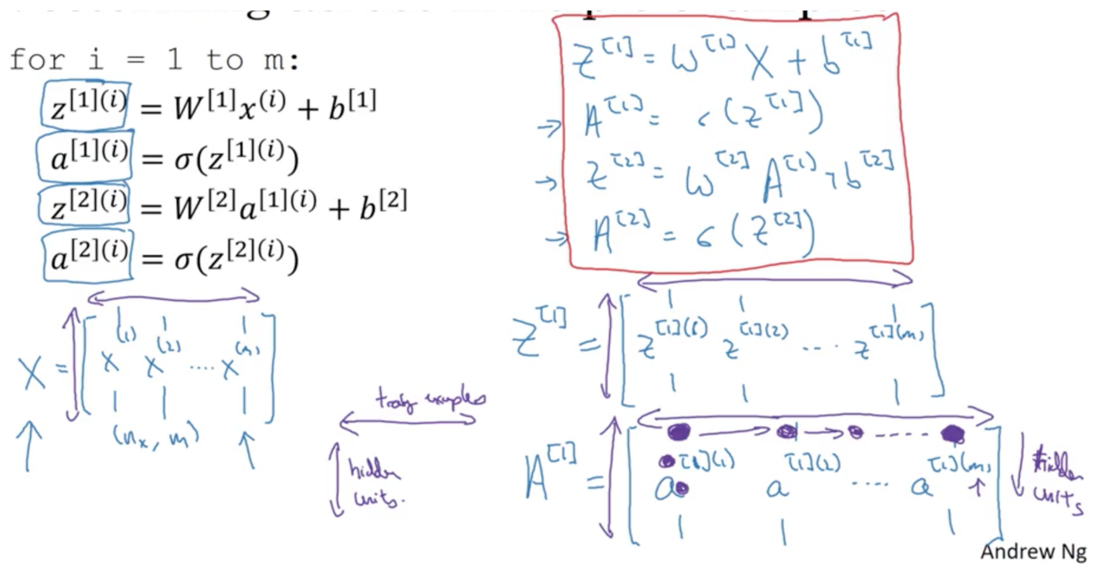

Neural Network Representation
$x_1$
$x_2$
$x_3$
$w^Tx+b$
$\sigma(z)$
$z$
$a$
$a=\hat{y}$
$z=w^Tx+b$
$a=\sigma(z)$
$z_1^{[1]}=w_1^{[1]T}x+b_1^{[1]}$
$a_1^{[1]}=\sigma(z_1^{[1]})$
$a_i^{[l]}$ ← layer
← node in layer.
$z_2^{[1]}=w_2^{[1]T}x+b_2^{[1]}$
$a_2^{[1]}=\sigma(z_2^{[1]})$

Figure 1. Each node in Neural Network does the same thing as the node in Logistic Regression. *Source: Coursera Deep Learning course*

Vectorization: $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$, $a^{[l]} = \sigma(z^{[l]})$.

**Vectorizing across multiple example**

$^{[l]\,(i)}$: layer l, training example i.



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

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

$X = [x^{(1)}\ x^{(2)} \dots x^{(m)}]$
$(n_x, m)$
training examples
hidden units
$Z^{[1]} = [z^{[1](1)}\ z^{[1](2)} \dots z^{[1](m)}]$
$A^{[1]} = [a^{[1](1)}\ a^{[1](2)} \dots a^{[1](m)}]$
hidden units
Andrew Ng

Figure 2. Vectorized implementation.
*Source: Coursera Deep Learning course*

**Explanation for Vectorized Implementation**

No notes.

**Activation functions**

Sigmoid is called as a **activation function**.

Another activation function works **mostly better** than Sigmoid function is **Hyperbolic Tangent** function (which is a shifted version of Sigmoid function).

It’s better because it crosses (0, 0) point, and tends to centering the data, so the mean of the data is close to 0, and this actually makes the learning for the next layer a little bit easier.

One exception is that when you use binary classification problem, and the output value is between 0 and 1, then you still use the Sigmoid activation function.

**The activation function can be different for different layers**: we can you tanh function for hidden layers and Sigmoid function for output layer.

One downside of tanh(x) and Sigmoid(x) function: if x is either too large or too small, the slope (derivative) of the function at x is close to 0, and this makes Gradient Descent slow.

Another popular choice of activation function: **Rectified Linear Unit (ReLU)**. The slope (derivative) of this function at x is equal to either 0 (x < 0) or 1 (x >= 0).

The fact that the slope of ReLU function equals 0 when x < 0, in practice it works just fine; because generally the z values of hidden units will be > 0.

Rules of thumps for choosing activation function:

Binary classification problem: Sigmoid function is a natural choice for output layer, for hidden layers: use ReLU function.

**Pros and Cons for activation function**

- Sigmoid: Never use this except for the output layer if you're doing binary classification problem, because we have:
- Tanh: Works mostly better than Sigmoid.
- ReLU: The most commonly used activation function, use this when you're not sure what to use.
- **Leaky ReLU**: Feel free to try.

**Why do you need non-linear activation functions**

Linear activation function/Identity activation function: .

If you use Linear activation function (don't use non-linear activation function), regardless of how many hidden layers you have, neural network just compute the output value as a linear function of input X. So you are not computing any interesting function even if you go deeper in the network.

There may be only 1 place that you use linear activation function: when you're doing machine learning on Regression problem (output value y is a real number).

Read more: Activation Function.

**Derivatives of activation function**

**Gradient Descent for (single layer) Neural Networks**

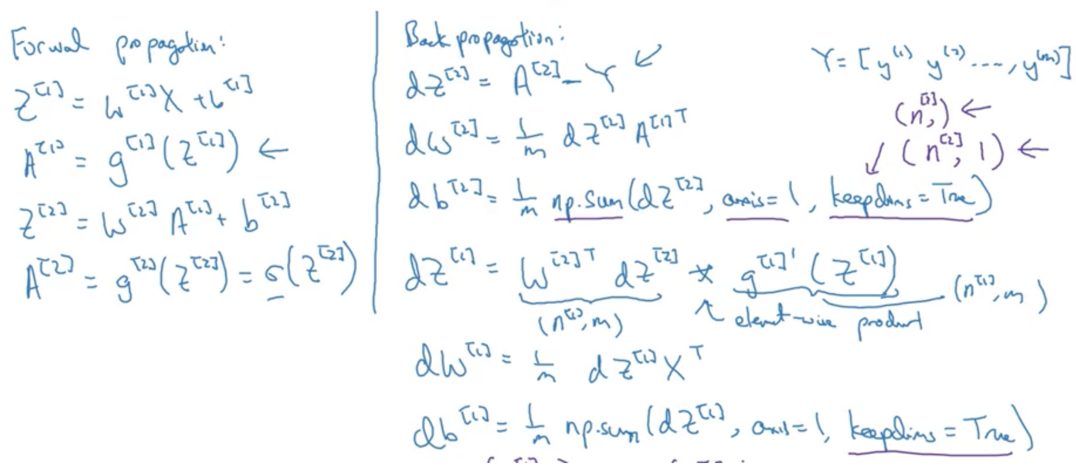
**Gradient Descent algorithm:**

Repeat {

- Feedforward: Compute predict for all training examples .
- Backpropagating: Calculate the derivatives .
- Perform gradient descent update:
  - .
  - .

}

**Formulas for computing derivatives**



Forward propagation:

$Z^{[1]} = W^{[1]}X + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$

$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$

Back propagation:

$dZ^{[2]} = A^{[2]} - Y$ $\quad Y = [y^{(1)}\ y^{(2)} \ldots, y^{(m)}]$

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

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

$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]},m)} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{(n^{[1]},m)}$ element-wise product

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

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

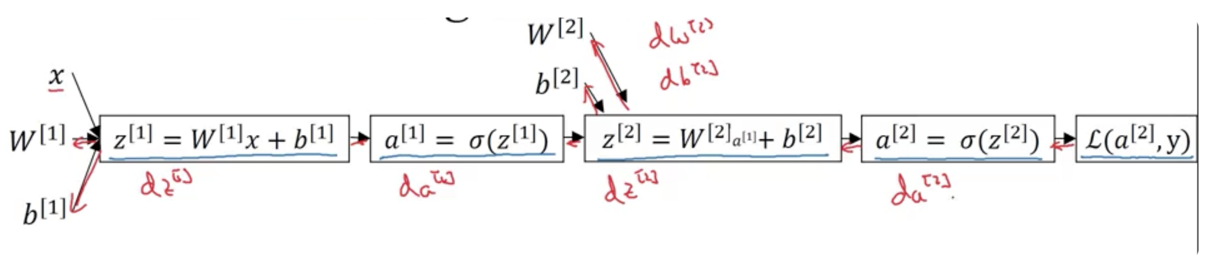
Figure 3. Formulas for computing derivatives.

*Source: Coursera Deep Learning course*

---

**Backpropagation intuition (optional)**

---



$x$
$W^{[1]}$
$b^{[1]}$
$z^{[1]} = W^{[1]}x + b^{[1]}$
$dz^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$da^{[1]}$
$W^{[2]}$
$b^{[2]}$
$dW^{[2]}$
$db^{[2]}$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$dz^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
$da^{[2]}$
$\mathcal{L}(a^{[2]}, y)$

Figure 4. Computation graph for 2 layer Neural Networks.

*Source: Coursera Deep Learning course*

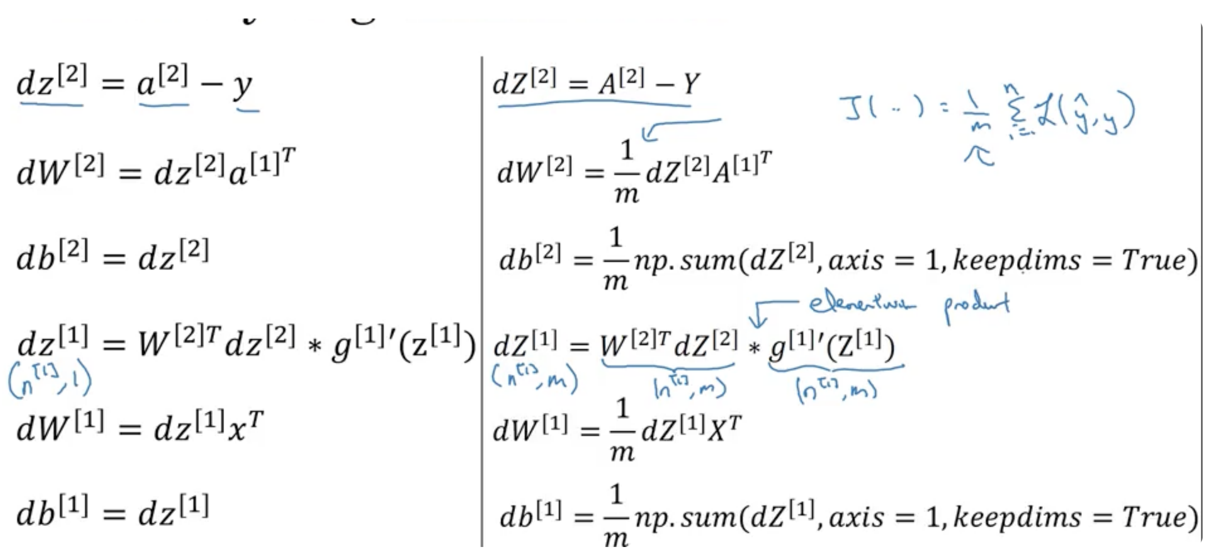
---

**Perform backpropagation (apply Chain rule)**

Similarly to .

All that is just the gradient for a single training example, we can vectorize to work with m training example:

---



$dz^{[2]} = a^{[2]} - y$ $\qquad dZ^{[2]} = A^{[2]} - Y$ $\qquad J(\ldots) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}, y)$

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

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

$\underbrace{dz^{[1]}}_{(n^{[1]},1)} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$ $\qquad \underbrace{dZ^{[1]}}_{(n^{[1]},m)} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]},m)} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{(n^{[1]},m)}$ elementwise product

$dW^{[1]} = dz^{[1]} x^T$ $\qquad dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$

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

Figure 5. Vectorization for m training examples.
*Source: Coursera Deep Learning course*

## Random Initialization

If you initialize weights (W, b) to 0, the hidden units will calculate exact the same function (this is bad because you want different hidden units to compute different functions).

Solution: Random Initialization:

- $w^{[l]}$ = np.random.rand((n, m)) * **0.01**, we usually prefer to initialize random small value, and it prevents the derivatives of activation function from being close to 0.
- $b^{[l]}$ = np.zeros((n, 1)), it's ok for b to be zero.

« Week 2 - Neural Networks Basics

Coursera Deep Learning Course 1 Week 4 notes: Deep neural networks »