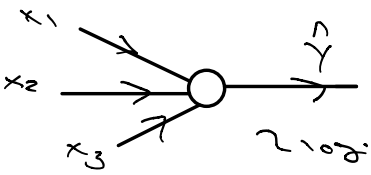
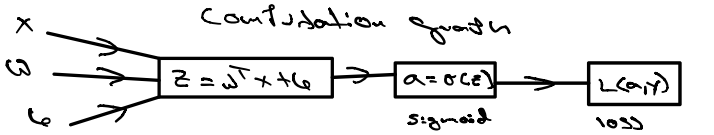
Week 3: Shallow Neural Networks

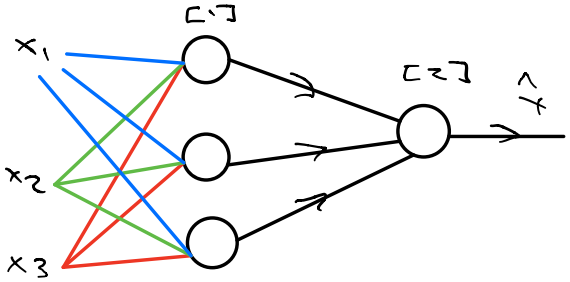
Overview



2 logistic regression



neural network



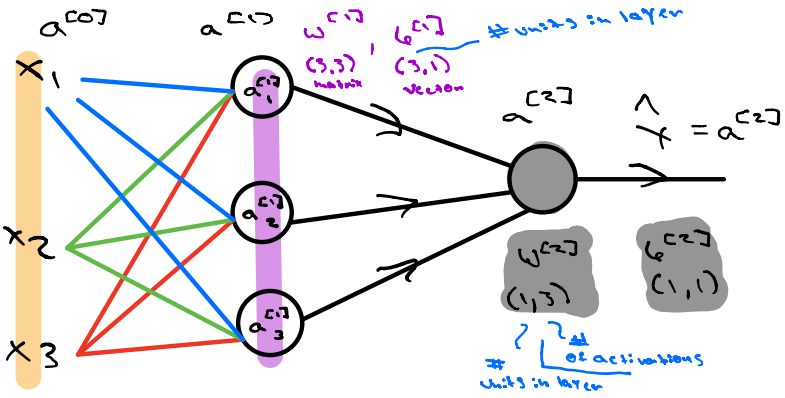
densely connected

z and a like calculations happen at each node/neuron

notation: $w^{[1]}$; represents the weights for the first layer in the neural net (superscript)

round brackets $(i) \rightarrow x^{(i)}$ are used to refer to training examples

Neural network Representation



input layer

hidden layer

output layer

↳ generates predicted value

- the name hidden layer comes from the fact that the true value of the nodes in this middle layer are not observed in the training set.
  - while they are observed for the inputs and outputs.

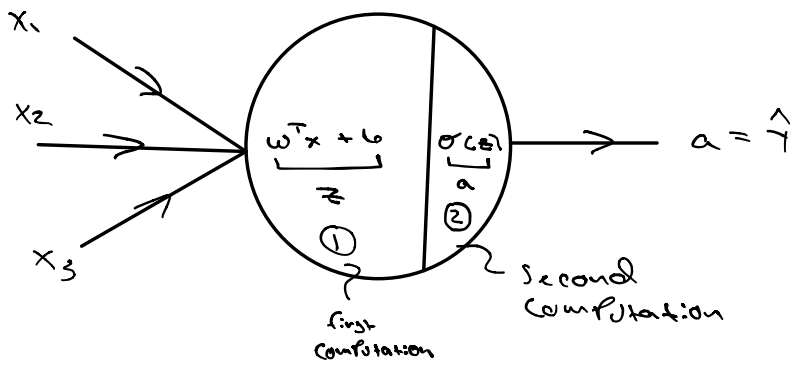
- input vector representation: vector X or $a^{[0]}$
  - a stands for activations. these refer to the values that layers pass on to subsequent layers.
    - $a^{[0]}$ is passed to the hidden layer

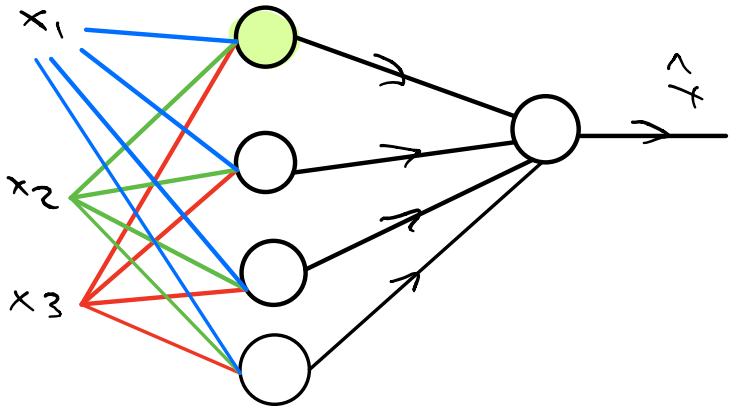
- the hidden layer generates a set of activations, $a^{[1]}$.
  Given that we have 3 nodes in this layer, $a^{[1]}$ is a 3×1 dimensional vector.
  - $a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix}$ — 3 hidden units in the layer

- the example above is a 2-layer neural network, as we don't count the input layer.
  - conventional usage definition

Computing a Neural Network's Output

Computing a neural network is like logistic regression but computed a lot of times.





- focusing on the first neuron of the first layer

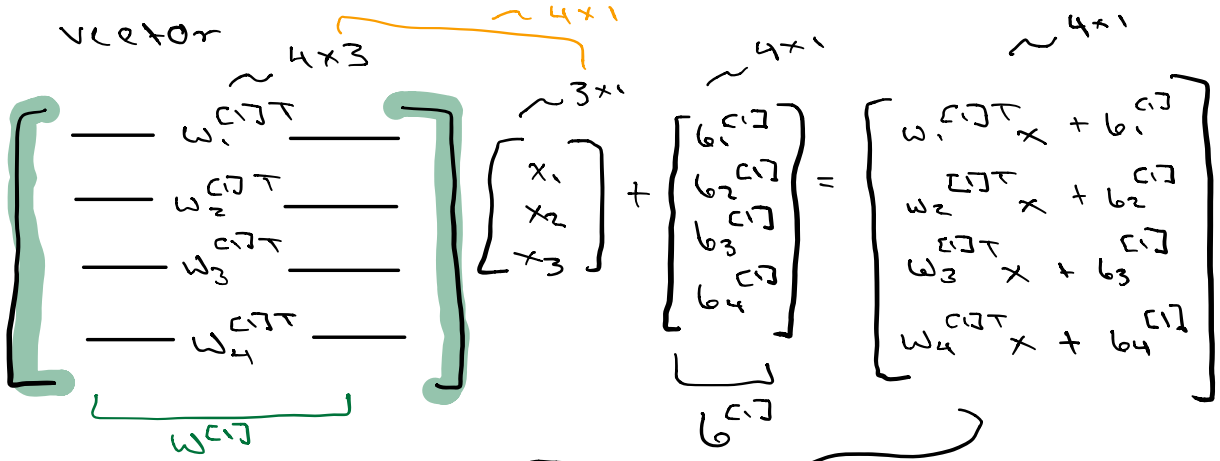
  Step 1: $z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$ (layer, node in layer)

  Step 2: $a_1^{[1]} = \sigma(z_1^{[1]})$

- these steps repeat for $z_2^{[1]} \rightarrow a_2^{[1]}$, $z_3^{[1]} \rightarrow a_3^{[1]}$ and $z_4^{[1]} \rightarrow a_4^{[1]}$

  ↳ this process is repetitive so, how can we vectorize?

W^T transforms a column vector into a row vector



- We have 4 logistic regression units in this matrix. Each of these logistic regression units has a parameter vector (w), we end up with a 4×3 matrix. 4 units, 3 activations per unit

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = z^{[1]}$$

$$\sigma(z^{[1]}) = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} = a^{[1]}$$

- We need a function that takes $z^{[1]}$ and applies the sigmoid function elements wise.

Simplified: (or $a^{[0]}$)

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

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

~ first layer

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

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

~ second layer

* this applies to one training example

Vectorizing across multiple examples

- to train with more training examples $(x^{(1)} \dots x^{(m)})$ we need to repeat the process summarized above.

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

training example, layer

- we would like to vectorize the computation of training on m examples as we don't want to loop for training example applying the same four equations above.

Standard for loop implementation:

for $i = 1$ to m:

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

to vectorize the training set we can stack them up in columns to create a $3 \times m$ matrix (for our example)

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

$(n_x, m) = (\text{features}, \text{examples})$

The vectorized implementation of the for loop is never,

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

* often case Z, W, X, A as they're all matrices. b is a column vector

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

~ similar for $Z^{[2]}$ & $A^{[2]}$

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

↓ hidden unit # per layer index

→ the horizontal index corresponds to training examples. The vertical index corresponds to different nodes in the layer being analyzed.

Justification for vectorized Implementation

$$W^{[1]} = \begin{bmatrix} — \\ — \\ — \end{bmatrix} \qquad W^{[1]} x^{(1)} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

Each row represents the weights connected to one neuron in the layer

When multiplied by a column vector of inputs (features), row • column, the first row $w_1^{[1]}$ is multiplied against the column $x^{(1)}$ and then summed up into one data point (this is done for all rows in $W^{[1]}$). Hence, each data point $x_1^{(1)}, x_2^{(1)}, x_3^{(1)} \dots x_n^{(1)}$ is scaled by a different $w^{[1]}$ value ($w_{11}, w_{21}, w_{31} \dots w_{n_1 1}$ (neuron in layer 1)).

Here, we are scaling the same data point by different weights depending on which neuron of the layer we are evaluating

└ this applies for any activation value not just $a^{[0]} = X$

- The sample calculation of one W row (one neuron in a layer) with one training example is given as:

the sum of a neuron's z with $b=0$

$(z_1^{[1]} = w_{1\_nx} \cdot x_{nx})$

$\hookrightarrow w_{11} \cdot x_1 + w_{12} \cdot x_2 + w_{13} \cdot x_3$

$(z_2^{[1]} = w_{2\_nx} \cdot x_{nx}$

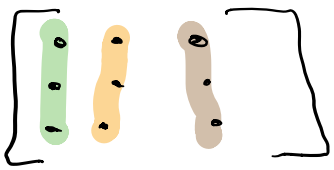
$\hookrightarrow w_{21} \cdot x_1 + w_{22} \cdot x_2 + w_{32} \cdot x_3$

When the examples are stacked vertically

Such that

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \dots \\ | & | & | \end{bmatrix}$$

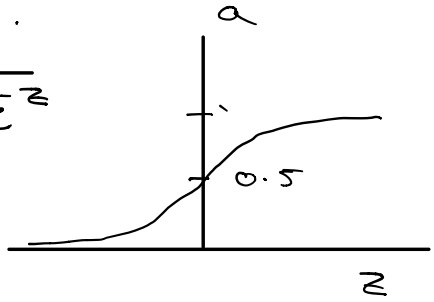
then $W^{[1]} X$ will be a matrix



where each column is the z for a different training example

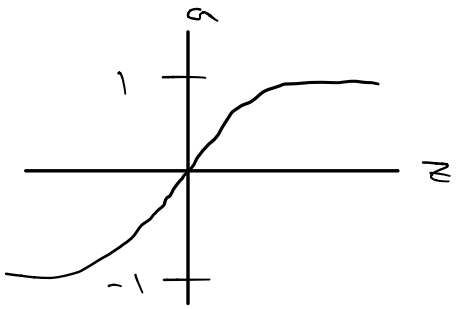
Activation functions

- So far we have been using the sigmoid activation function, but sometimes, other choices can work much better.
  - Sigmoid: $a = \frac{1}{1 + e^{-z}}$



- Instead of $a = \sigma(z)$ we could have $a = g(z)$, where g could be other linear functions which are not the sigmoid.

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



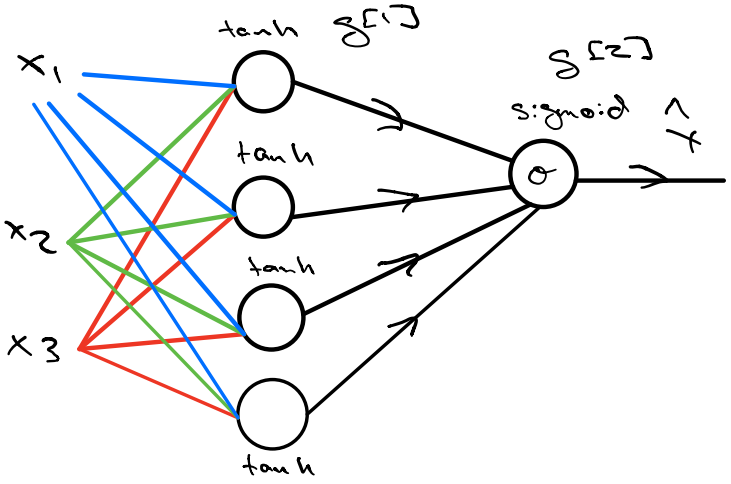
  - using tanh instead of the sigmoid for the hidden units almost always works better because with values $[1, -1]$, the mean of

the activations that come out of the layer are closer to zero.

↳ the data is centered around zero instead of 0.5 making learning a bit easier.

↳ the one exception for using the sigmoid function is the output layer because y is either zero or one and therefore it makes sense for $\hat{y}$ to be a number in this range. (in binary classification)

- to indicate that the activation functions might be different for different layers, we utilize another superscript $(g^{[1]}(z^{[1]})$

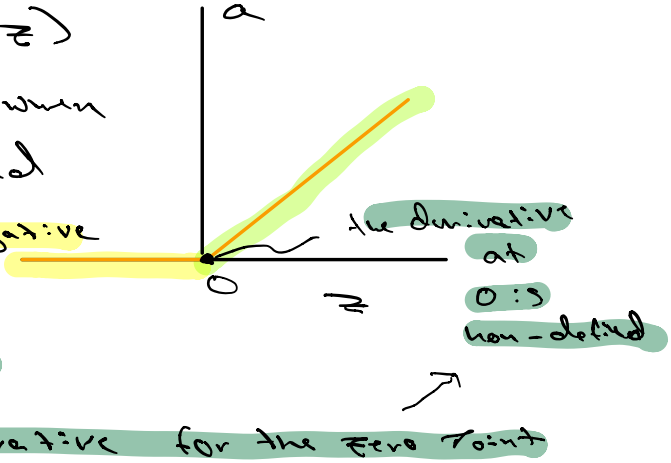


- one of the major downsides to both the sigmoid and tanh is that if z is either very large or very small, then the gradient of these

functions becomes very small.

- ↳ close to zero, slowing down gradient descent.

- Given this drawback, the rectified linear unit is used as an alternative (RELU)
  - ↳ RELU: $a = \max(0, z)$
    - ↳ the derivative is 1 when z is positive and 0 when z is negative
      - ↳ but we can use either one or zero derivative for the zero point



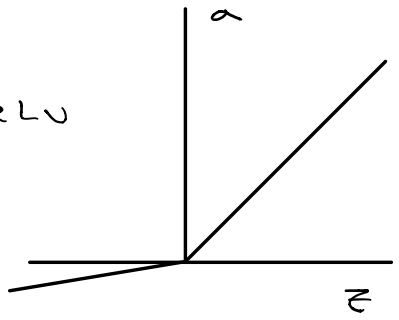
- One disadvantage of the ReLU is that its derivative is 0 for negative. In practice this isn't an issue, but a alternative called the Leaky ReLU is sometimes used.

Rules of thumb for choosing Activation functions

∴ Binary classification: sigmoid at output layer is the natural choice

- ↳ for all other units the RELU is used (in the hidden layer) ... rarely tanh

- Leaky ReLU:
  - usually works better than ReLU
    - $a = \max(0.01 \cdot z, z)$



Why do we need non-linear activation functions

- to compute interesting functions, we require non-linear activation functions

- if we set $g(z) = z$, we are computing our prediction as a linear function of our input features
  - $a^{[1]} = z^{[1]} = W^{[1]} x + b^{[1]}$
  - $a^{[2]} = z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

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

$$= \underbrace{(W^{[2]} W^{[1]})}_{W'} x + \underbrace{(W^{[2]} b^{[1]} + b^{[2]})}_{b'}$$

$$= W' x + b'$$

* using linear activations (identity activations), the neural network will only output a linear function of the input
  - stacking more layers in a deep neural net won't have any effect as we are always calculating a linear function of the

inputs.

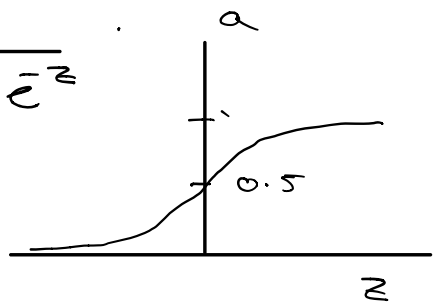
- a linear hidden unit is useless because the composition of two linear functions is itself a linear function (theorem of superposition).

- We might only use linear activations if we are doing machine learning on a regression problem.
  - the $\hat{y}$ will be a real number in this case so it's sensible.
  - but the hidden units must be non linear (such as ReLU or tanh)

* linear units are sometimes used for compression but not discussed in the course.

Derivatives of activation functions

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



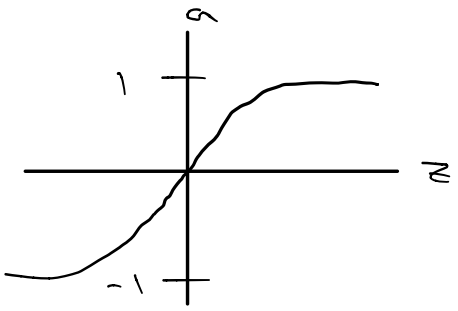
$$\frac{d}{dz} g(z) = a(1-a)$$

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

the advantage of this formula is that if we've already computed the the activation (a),

- it's very easy to compute the slope as $a(1-a)$.

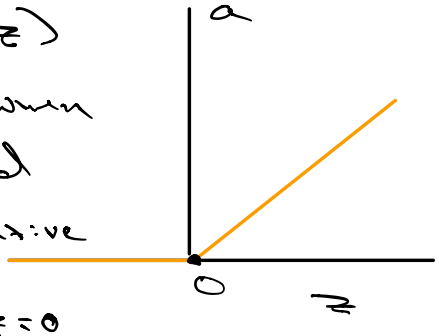
- tanh function (hyperbolic tan)
  - $a = \tanh(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$



$$\frac{d}{dz} g(z) = g'(z) = 1 - (\tanh(z))^2$$

  - this formula again can be used to quickly compute the derivative of tanh

- RELU: $a = \max(0, z)$
  - the derivative is 1 when z is positive and 0 when z is negative
    - undefined if $z = 0$



Gradient Descent for Neural Networks

- a neural network with a single hidden layer has parameters $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$

$n_x = n^{[0]}$ ~ input features, $n^{[1]}$ ~ hidden units, $n^{[2]} = 1$ ~ output layer

- $W^{[1]} = (n^{[1]}, n^{[0]})$, $b^{[1]} = (n^{[1]}, 1)$
- $W^{[2]} = (n^{[2]}, n^{[1]})$, $b^{[2]} = (n^{[2]}, 1)$

$n^{[1]}$: number of neurons in layer; $n^{[0]}$: connections to previous layer

- for binary classification, we have a cost function

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

* $\hat{y} = a^{[2]}$ — loss function average.

* when we have a neural network it's better to initialize all the parameters randomly (instead than set 0).

- Gradient descent:

Repeat {

- Compute predictions ($\hat{y}^{(i)}$, for $i = 1$ through m)
- Compute derivatives
  - $dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$ ... for $W^{[2]}$ and $b^{[2]}$
- Update parameters:
  - $W^{[1]} = W^{[1]} - \alpha \cdot dW^{[1]}$
  - $b^{[1]} = b^{[1]} - \alpha \cdot db^{[1]}$
  - $W^{[2]} = W^{[2]} - \alpha \cdot dW^{[2]}$
  - $b^{[2]} = b^{[2]} - \alpha \cdot b^{[2]}$

}

- now do we compute these partial derivatives?
  - forward propagation:

Similar to logistic regression

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

~ sigmoid for binary classification

g term backtrack in sigmoid term

Back propagation:

ground truth ~ row vector

$Y_{(1\times m)} = [y^{(1)} \dots y^{(m)}]$

$$dz^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} \cdot dz^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \cdot np.sum(dz^{[2]}, axis=1, keepdims=True)$$

? horizontal sum $\to (n^{[2]}, 1)$

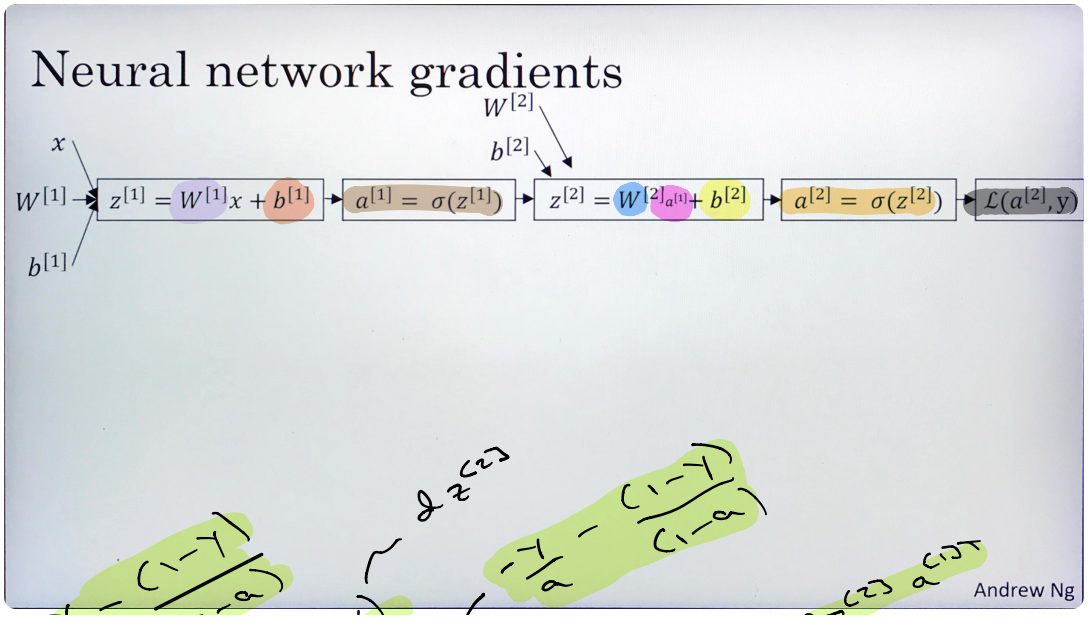
$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$$

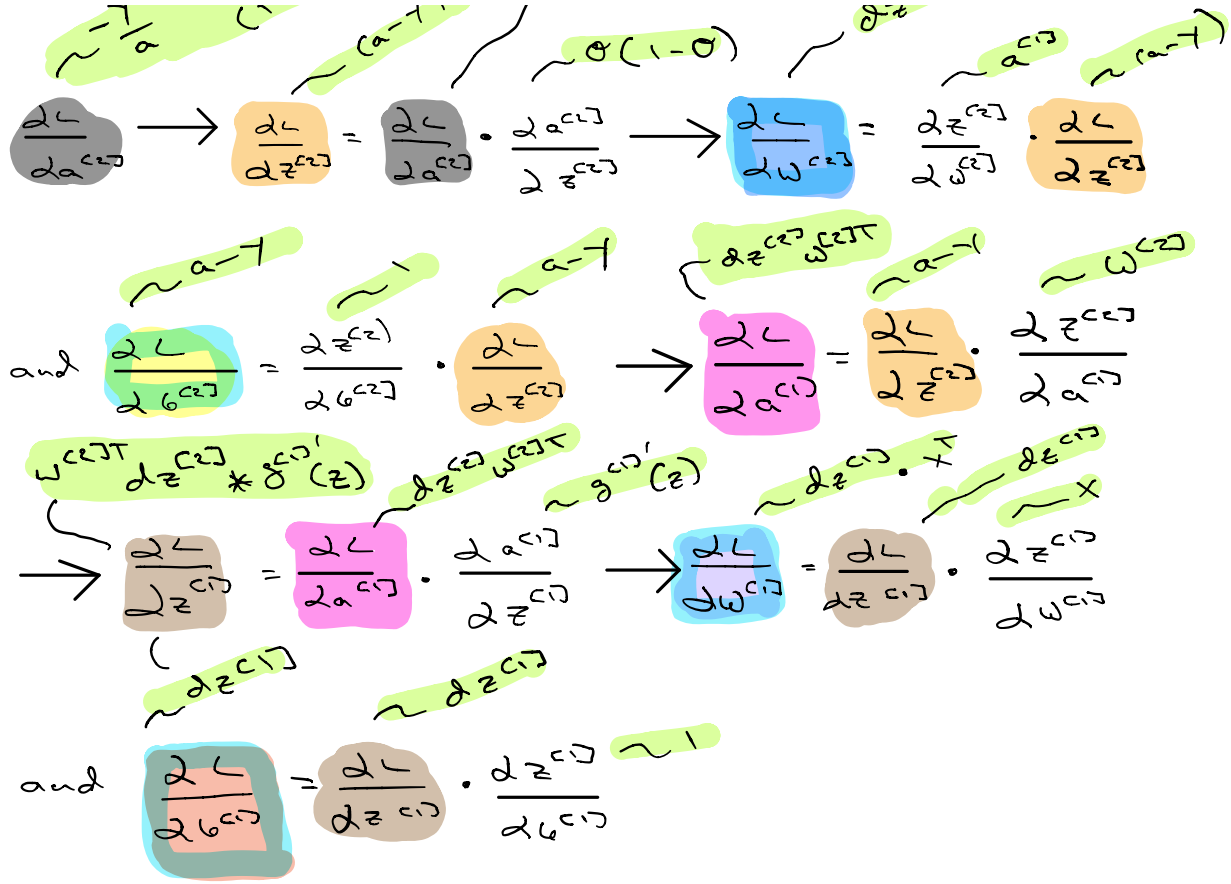
$(n^{[1]}, m)$ $(n^{[1]}, m)$

element wise product

$$dW^{[1]} = \frac{1}{m} \cdot dz^{[1]} X^T \sim A^{[0]}$$
$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims=True)$$

Computational Graph of 1-layer neural network





- Vectorized Implementation of Back Propagation (Column vectors)

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

$$J(\cdot) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}, y)$$

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

$A^{[1]T}$ ← transposed row vector

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

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

$*$ ← element-wise product ← each element with one other element in the same place in the other matrix

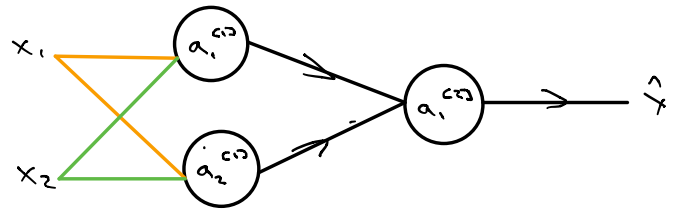
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

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

Andrew Ng

Random Initialization

- in a neural network, if we initialize the parameters to zero and then apply gradient descent, it won't work.



$n^{[0]} = 2 \qquad n^{[1]} = 2$

therefore, $w^{[1]} = (2, 2)$

~ this is a problem

~ this is fine

$$w^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- the problem with the initialization to zero is that for any given example, $a_1^{[1]}$ and $a_1^{[2]}$ are equal.
  - ↳ both hidden units will compute the same function
  - ↳ when computing each propagation, $dz^{[1]} = dz^{[2]}$ by symmetry.
    - ↳ both hidden units are computing the same function
    - ↳ through induction, it's proved that after multiple iterations of gradient descent, these will be computing the same function.
      - ↳ $dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix} \rightarrow w^{[1]} = w^{[1]} - \alpha\, dw^{[1]}$ all rows will be equal after each update. The point in having more than

one neuron) (symmetry breaking problem)

- the solution to this is to initialize parameters randomly.
  - $W^{[1]} = np.random.randn((2,2)) * 0.01$
    - gaussian random variable
  - $b^{[1]} = np.zeros((2,1))$

* we multiply by a small # because we want to have activations that are different when we input the values into our sigmoid or tanh (the sloped area provides better/faster performance in gradient descent).
  - a saturated sigmoid function slows down learning

Additional

* In general, the # of neurons in the previous layer provides the # of columns, while the # of neurons in the current layer gives us the # of rows of the weight matrix...

for the activation matrix, the row gives the neuron in a specific layer, while the the columns give different examples.

The resulting $a^{[l]}$ is a combination of the activations for the neurons in a layer (rows) with the activation for different examples (columns).