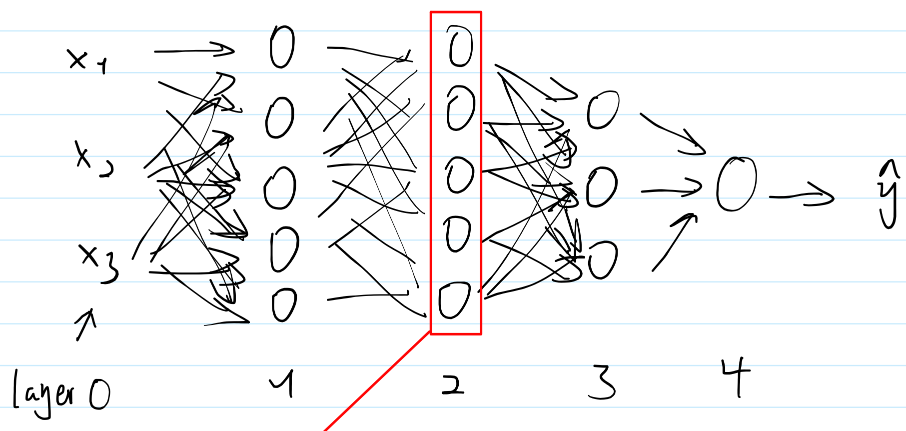
Week4 - Deep Neural Networks

Samstag, 18. Juli 2020 09:24

Deep NN with 4 layers



$L = 4$ (# layer)

$n^{[l]}$ (# units in l layer)

$n^{[1]} = n^{[2]} = 5$

$n^{[3]} = 3 \qquad n^{[4]} = 1$

$n^{[0]} = n_x = 3$

$a^{[l]}$ = activation in l-th layer $\qquad \hat{y} = a^{[L]} \qquad x = a^{[0]}$

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \qquad a^{[l]} = g^{[l]}(z^{[l]})$

What happens in layer l:

Parameters: $W^{[l]}$ $b^{[l]}$

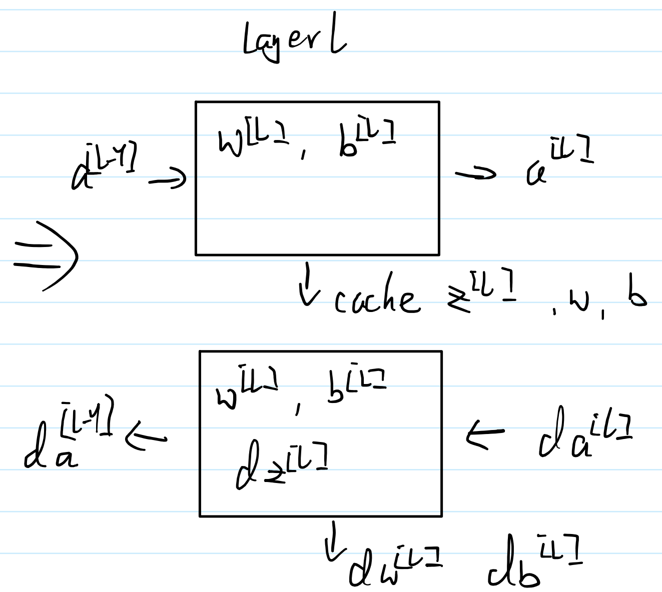
Forward: Input $a^{[l-1]}$ output $a^{[l]}$

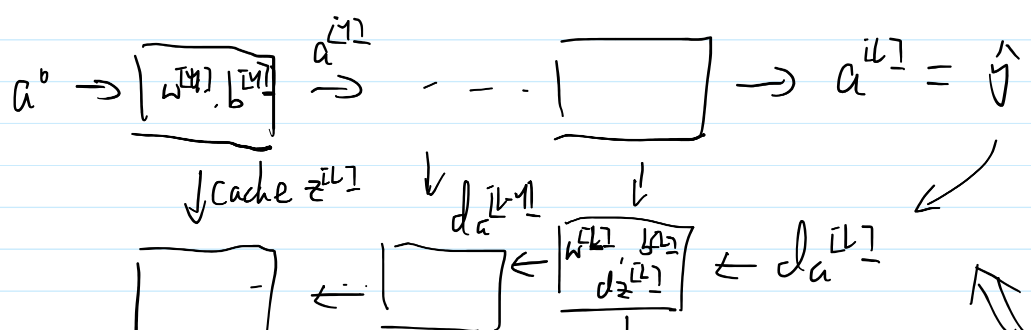
$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \Rightarrow$ cache $z^{[l]}$

$a^{[l]} = g^{[l]}(z^{[l]})$

Backward: Input $\left\{ da^{[l]}, \text{cache } z^{[l]} \right.$, output $\left\{ da^{[l-1]}, dW^{[l]}, db^{[l]} \right.$

$\Longrightarrow$





Cache $\left\{ z^{[l]}, W^{[l]}, b^{[l]} \right.$

$W^{[L]}, b^{[L]}$, $dz^{[L]}$ ← $da^{[L]}$ One Iteration

$dw^{[1]}$ $db^{[1]}$ → update

$dw^{[L]}$ $db^{[L]}$ → update

Backward Propagation

① $dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$

The three outputs $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$ are computed using the input $dZ^{[l]}$ .Here are the formulas you need:

② $$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

③ $$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

④ $$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

Hyperparameters:

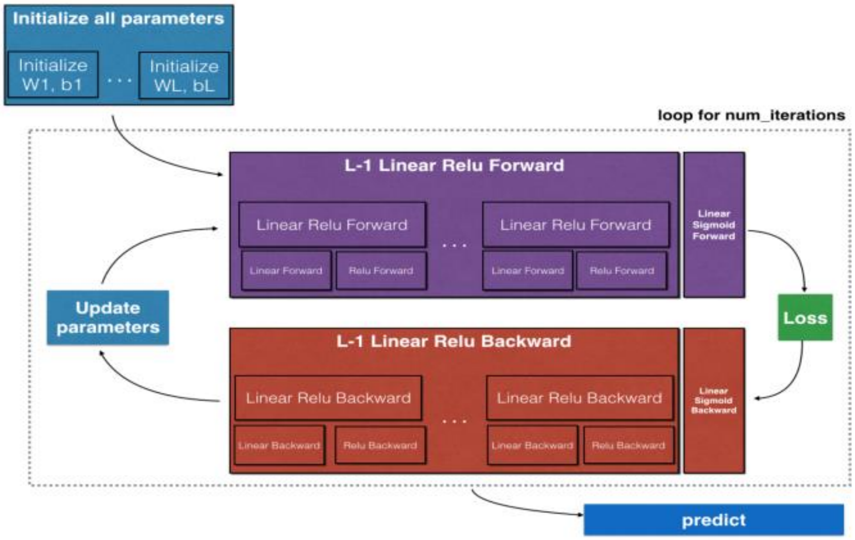
α
Iteration
L
$n^{[L]}$
$g^{[L]}$ ( )

⇒ Idea → code → Experiment

Applying Deep Learning in a empirical process

Take away from the assignment:

Deep Neural Network Step by Step



basic outline:

- Initialize the parameters for a two-layer network and for an L -layer neural network.
- Implement the forward propagation module (shown in purple in the figure).
  - Complete the LINEAR part of a layer's forward propagation step (resulting in Z[l]). Then into activation step, which gets a new [LINEAR->ACTIVATION] forward function.
  - Stack the [LINEAR->RELU] forward function L-1 time (for layers 1 through L-1) and add a [LINEAR->SIGMOID] at the end (for the final layer L). -> L_model_forward function.
- Compute the loss.
- Implement the backward propagation module (denoted in red in the figure).
  - Complete the LINEAR part of a layer's backward propagation step.
  - Compute gradient of the ACTIVATE function (relu_backward/sigmoid_backward)
  - Combine the previous two steps into a new [LINEAR->ACTIVATION] backward function.
  - Stack [LINEAR->RELU] backward L-1 times and add [LINEAR->SIGMOID] backward in a new L_model_backward function
- Finally update the parameters.

1. Initialization of parameters of a two layer NN (one hidden layer)

```
def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer
    Returns:
    parameters -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

2. Initialization of L layer NN

E.g. X has shape of (12288, 209) -- m = 209 examples

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

The model's structure is [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID.
It is convient to store # units of layer l in a list layer_dims

For L == 1:

```
if L == 1:
    parameters["W" + str(L)] = np.random.randn(layer_dims[1], layer_dims[0]) * 0.01
    parameters["b" + str(L)] = np.zeros((layer_dims[1], 1))
```

```
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- list containing the dimensions of each layer in our network
    Returns:
    parameters -- dictionary containing parameters "W1", "b1", ..., "WL", "bL":
                    Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """
    parameters = {}
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

    return parameters
```

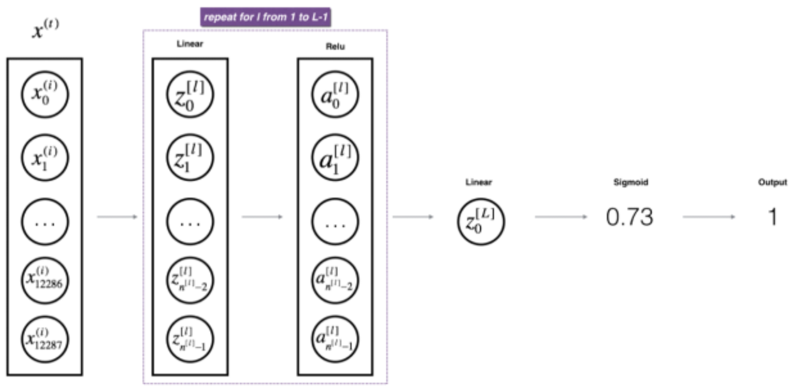
3. Linear forward

```
def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.
    Arguments:
    A -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function
    cache -- a tuple containing "A", "W" and "b" ; stored for computing the backward pass
    """

    Z = np.dot(W, A) + b
    cache = (A, W, b)

    return Z, cache
```

4. Linear activation forward

```
def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as string: "sigmoid" or "relu"

    Returns:
    A -- the output of the activation function
    cache -- a tuple containing "linear_cache" and "activation_cache" stored for computing the backward pass
    """

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z) # activation_cache contains z
    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)  # linear cache = (A, W, b) A is activation from previous layer
     #activation cache = Z current Z to get A

    return A, cache
```

5. L layer model




```
def L_model_forward(X, parameters):
    """
    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_activation_forward() (there are L-1 of them, indexed from 0 to L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2                  # number of layers in the neural network

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):  # 1:(L-1)
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)], parameters['b' + str(l)], "relu")
        caches.append(cache)

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], parameters['b' + str(L)], "sigmoid")
    caches.append(cache)

    return AL, caches
```

6. Cost Function

7. Linear Backward

```
def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward propagation in the current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1/m * np.dot(dZ, A_prev.T)
    db = 1/m * np.sum(dZ, axis = 1, keepdims = True)   # (n_l,1) sum over all m examples
    dA_prev = np.dot(W.T, dZ)

    return dA_prev, dW, db
```

8. Linear activation backward

```
def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.
    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z'
    Returns:
    dZ -- Gradient of the cost with respect to Z
    """
    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    return d

def sigmoid_backward(dA, cache):
    """
    Implement the backward propagation for a single SIGMOID unit.
    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z'
    Returns:
    dZ -- Gradient of the cost with respect to Z
    """
    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    return dZ

def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache)
    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    linear_cache, activation_cache = cache
```

```
    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)  # activation cache = Z_l
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

9. Model Backward

```
def L_model_backward(AL, Y, caches):
    """
    Arguments:
    AL -- output of the forward propagation (L_model_forward())
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu" (it's caches[l], for l in range(L-1) i.e l = 0...L-2)
                the cache of linear_activation_forward() with "sigmoid" (it's caches[L-1])

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}   # store derivatives
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))  # input for backward propagation

    # L-1th layer (SIGMOID -> LINEAR) gradients.
    current_cache = caches[-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dAL, current_cache,"sigmoid")

    # Loop from l=L-2 to l=0
    for l in reversed(range(L-1)):  # l = L-2 : 0
        # lth layer: (RELU -> LINEAR) gradients.
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l+1)], current_cache,"relu")
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

10. Update

```
    def update_parameters(parameters, grads, learning_rate):

        L = len(parameters) // 2 # number of layers in the neural network

        # Update rule for each parameter. Use a for loop.
        for l in range(L):
            parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate * grads["dW" + str(l+1)]
            parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate * grads["db" + str(l+1)]
        return parameters
```

For any application:
General methodology:
1. Initialize parameters / Define hyperparameters
2. Loop for num_iterations:
   a. Forward propagation
   b. Compute cost function
   c. Backward propagation
   d. Update parameters (using parameters, and grads from backprop)
4. Use trained parameters to predict labels