Reference

Coursera deep learning series by Andrew NG.

Notation:

- Superscript $[l]$ denotes a quantity associated with the l^{th} layer.
  - Example: $a^{[L]}$ is the L^{th} layer activation. $W^{[L]}$ and $b^{[L]}$ are the L^{th} layer parameters.
- Superscript (i) denotes a quantity associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example.
- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a^{[l]}_i$ denotes the i^{th} entry of the l^{th} layer's activations).

1 - Packages

In [13]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
from testCases import *
from dnn_utils import *


%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

2 - Outline of the Assignment

- 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 below).
    - Complete the LINEAR part of a layer's forward propagation step (resulting in $Z^{[l]}$).
    - ACTIVATION function (relu/sigmoid) is provided.
    - Combine the previous two steps into 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). This gives you a new L_model_forward function.
- Compute the loss.
- Implement the backward propagation module (denoted in red in the figure below).
    - Complete the LINEAR part of a layer's backward propagation step.
    - Gradient of the ACTIVATE function (relu_backward/sigmoid_backward) is provided.
    - 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.

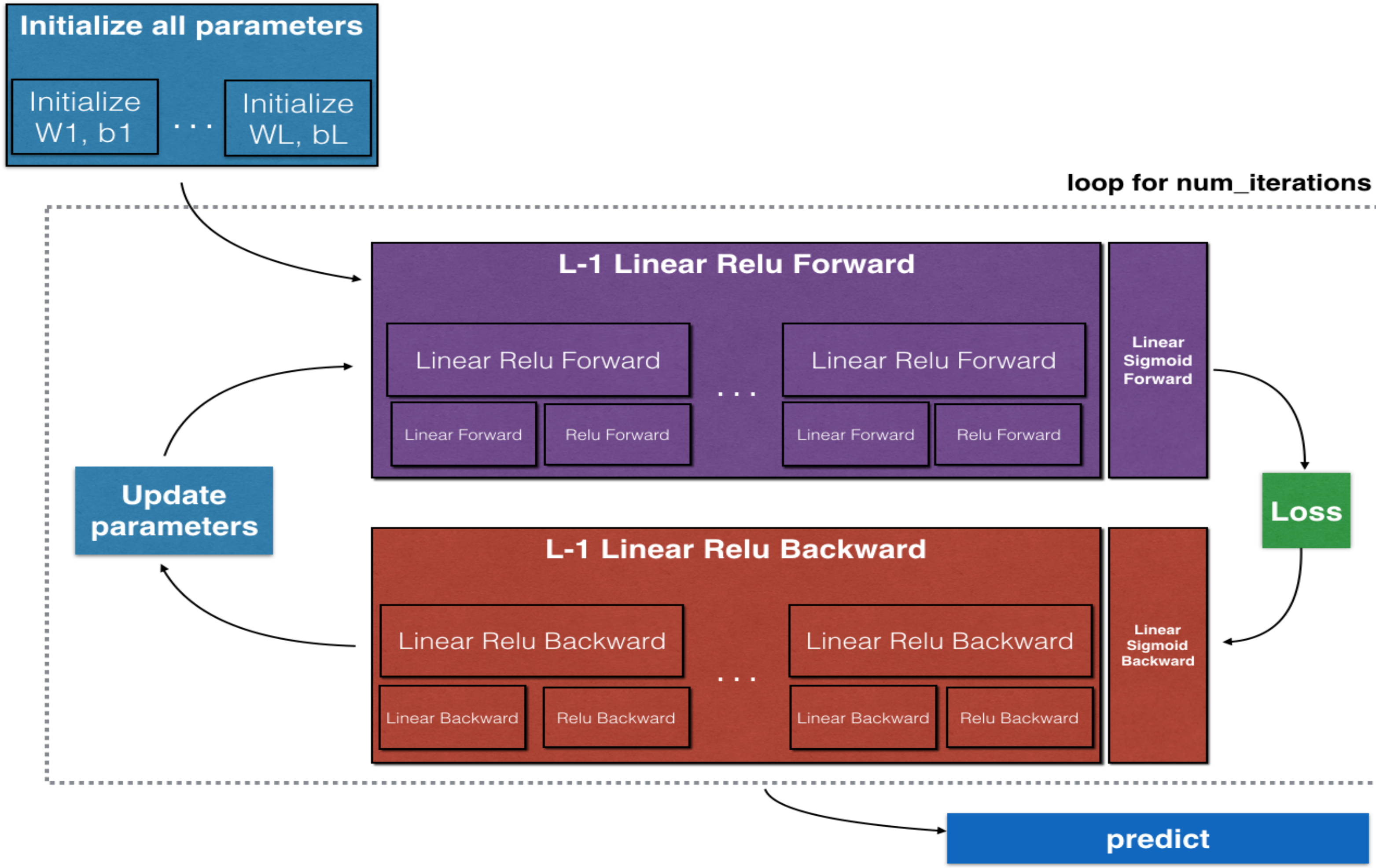

Figure 1

Note that for every forward function, there is a corresponding backward function. That is why at every step of your forward module you will be storing some values in a cache. **The cached values are useful for computing gradients**. In the backpropagation module you will then use the cache to calculate the gradients.

3 - Initialization

3.1 - 2-layer Neural Network

Instructions:

- The model's structure is: *LINEAR -> RELU -> LINEAR -> SIGMOID*.
- Use random initialization for the weight matrices. Use `np.random.randn(shape)*0.01` with the correct shape.
- Use zero initialization for the biases. Use `np.zeros(shape)` .

In [14]:
```
def initialize_parameters(n_x, n_h, n_y):
    np.random.seed(1)

    W1 = np.random.randn(n_h, n_x) * 0.01 #random number for normal distribution
    b1 = np.zeros(shape=(n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros(shape=(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
```

In [15]:
```
parameters = initialize_parameters(2,2,1)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[ 0.01624345 -0.00611756]
 [-0.00528172 -0.01072969]]
b1 = [[0.]
 [0.]]
W2 = [[ 0.00865408 -0.02301539]]
b2 = [[0.]]
```

3.2 - L-layer Neural Network

Recall that $n^{[l]}$ is the number of units in layer l. Thus for example if the size of our input X is $(12288, 209)$ (with $m = 209$ examples) then: **Normally, the input is not regarded as an official neural network layer. The first layer indexed as 1 is actually the second layer in a schematic figure. However, the table below takes the input layer as L = 1, which is different from the convention**.

	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	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

Exercise: Implement initialization for an L-layer Neural Network.

Instructions:

- The model's structure is *[LINEAR -> RELU]* $\times$ *(L-1) -> LINEAR -> SIGMOID*. I.e., it has $L-1$ layers using a ReLU activation function followed by an output layer with a sigmoid activation function.
- Use random initialization for the weight matrices. Use `np.random.rand(shape) * 0.01` .
- Use zeros initialization for the biases. Use `np.zeros(shape)` .
- We will store $n^{[l]}$, the number of units in different layers, in a variable `layer_dims` . For example, the `layer_dims` for the "Planar Data classification model" from last week would have been [2,4,1]: There were two inputs, one hidden layer with 4 hidden units, and an output layer with 1 output unit. Thus means `W1` 's shape was (4,2), `b1` was (4,1), `W2` was (1,4) and `b2` was (1,1). Now you will generalize this to L layers!
- Here is the implementation for $L = 1$ (one layer neural network). It should inspire you to implement the general case (L-layer neural network).

```
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))
```

Be careful the different implementations on whether taking the input layer as $L = 0$ and $L = 1$.

In [16]:
```
def initialize_parameters_deep(layer_dims):
    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)   # number of layers in the network

    for l in range(1, L): # Becareful l starts from 1 but not 0.
        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))

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l - 1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters
```

In [17]:
```
parameters = initialize_parameters_deep([5,4,3])
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[ 0.01788628  0.0043651   0.00096497 -0.01863493 -0.00277388]
 [-0.00354759 -0.00082741 -0.00627001 -0.00043818 -0.00477218]
 [-0.01313865  0.00884622  0.00881318  0.01709573  0.00050034]
 [-0.00404677 -0.0054536  -0.01546477  0.00982367 -0.01101068]]
b1 = [[0.]
 [0.]
 [0.]
 [0.]]
W2 = [[-0.01185047 -0.0020565   0.01486148  0.00236716]
 [-0.01023785 -0.00712993  0.00625245 -0.00160513]
 [-0.00768836 -0.00230031  0.00745056  0.01976111]]
b2 = [[0.]
 [0.]
 [0.]]
```

4 - Forward propagation module

4.1 - Linear Forward

Now that you have initialized your parameters, you will do the forward propagation module. You will start by implementing some basic functions that you will use later when implementing the model. You will complete three functions in this order:

- LINEAR
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- [LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID (whole model)

The linear forward module (vectorized over all the examples) computes the following equations:

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

where $A^{[0]} = X$.

In [18]:
```
#A, W, b = linear_forward_test_case()
import numpy as np
W = np.array([[1.0,1.3],[2.0,3.3],[4.0,5.3]])
b = np.array([[2.7],[3.5],[2.4]])
A = np.array([[1.0,2.8,3.5,2.9],[2.2,3.9,4.8,3.3]])
Z, linear_cache = linear_forward(A, W, b)
print("Z = " + str(Z))
```

```
Z = [[ 6.56 10.57 12.44  9.89]
 [12.76 21.97 26.34 20.19]
 [18.06 34.27 41.84 31.49]]
```

4.2 - Linear-Activation Forward

Use two activation functions:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$. We have provided you with the `sigmoid` function. This function returns **two** items: the activation value " `a` " and a " `cache` " that contains " `Z` " (it's what we will feed in to the corresponding backward function).

  ```
  A, activation_cache = sigmoid(Z)
  ```

- **ReLU**: The mathematical formula for ReLu is $A = RELU(Z) = max(0, Z)$. We have provided you with the `relu` function. This function returns **two** items: the activation value " `A` " and a " `cache` " that contains " `Z` " (it's what we will feed in to the corresponding backward function).

```
        A, activation_cache = relu(Z)
```

For more convenience, group two functions (Linear and Activation) into one function (LINEAR->ACTIVATION). Hence, implement a function that does the LINEAR forward step followed by an ACTIVATION forward step.

In [19]:
```
def linear_activation_forward(A_prev, W, b, activation):

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)
    # Note the difference of linear_cache and activation_cache: (A,W,b) and Z respectively.
    # The cache here returned includes both linear_cache and activation_cache.

    return A, cache
```

In [20]:
```python
#A_prev, W, b = linear_activation_forward_test_case()
import numpy as np
W = np.array([[1.0,1.3],[2.0,3.3],[4.0,5.3]])
b = np.array([[2.7],[3.5],[2.4]])
A_prev = np.array([[1.0,2.8,3.5,2.9],[2.2,3.9,4.8,3.3]])

A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activation = "sigmoid")
print("With sigmoid: A = " + str(A))

A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activation = "relu")
print("With ReLU: A = " + str(A))
```

```
With sigmoid: A = [[0.99858612 0.99997433 0.99999604 0.99994932]
 [0.99999713 1.         1.         1.        ]
 [0.99999999 1.         1.         1.        ]]
With ReLU: A = [[ 6.56 10.57 12.44  9.89]
 [12.76 21.97 26.34 20.19]
 [18.06 34.27 41.84 31.49]]
```

Note: In deep learning, the "[LINEAR->ACTIVATION]" computation is counted as a single layer in the neural network, not two layers.

4.3 L-Layer Model

For even more convenience when implementing the L-layer Neural Net, you will need a function that replicates the previous one (`linear_activation_forward` with RELU) $L-1$ times, then follows that with one `linear_activation_forward` with SIGMOID.

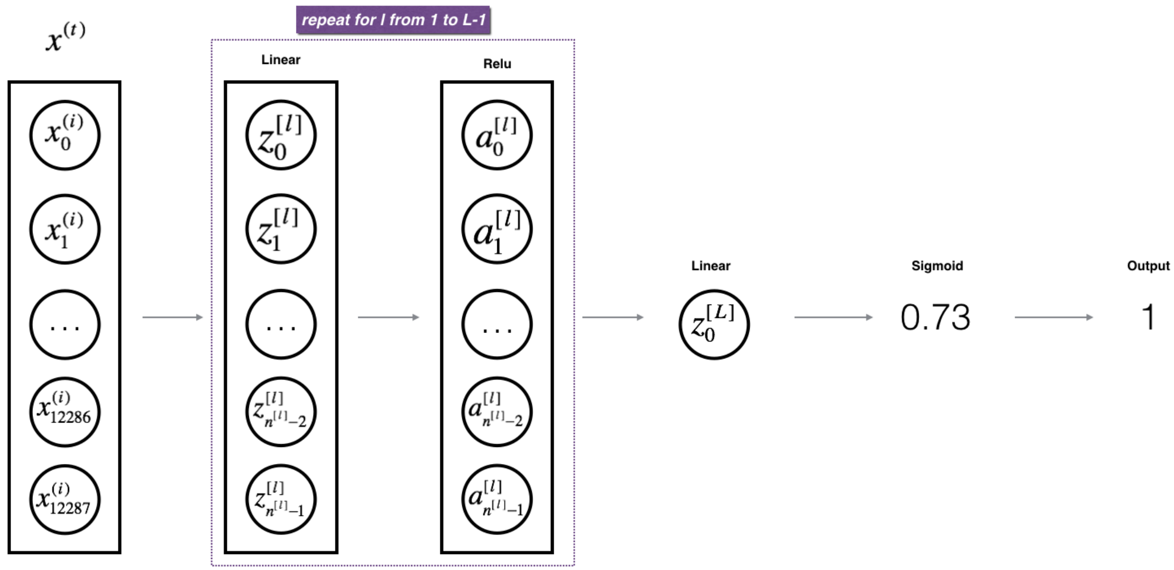

Figure 2 : *[LINEAR -> RELU]* $\times$ *(L-1) -> LINEAR -> SIGMOID* model

Exercise: Implement the forward propagation of the above model.

Instruction: In the code below, the variable `AL` will denote $A^{[L]} = \sigma(Z^{[L]}) = \sigma(W^{[L]}A^{[L-1]} + b^{[L]})$. (This is sometimes also called `Yhat` , i.e., this is $\hat{Y}$.)

Tips:

- Use the functions you had previously written
- Use a for loop to replicate [LINEAR->RELU] (L-1) times
- Don't forget to keep track of the caches in the "caches" list. To add a new value `c` to a `list` , you can use `list.append(c)` .

In [21]:
```
def L_model_forward(X, parameters):

    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):
        A_prev = A
        A, cache = linear_activation_forward(A_prev,
                                             parameters['W' + str(l)],
                                             parameters['b' + str(l)],
                                             activation='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)],
                                          activation='sigmoid')
    caches.append(cache)

    assert(AL.shape == (1, X.shape[1]))

    return AL, caches
```

In [22]:

```
import numpy as np
W1 = np.array([[1.0,1.3],[2.0,3.3],[4.0,5.3]])
b1 = np.array([[2.7],[3.5],[2.4]])
W2 = np.array([[2.0,3.3,4.3]])
b2 = np.array([[3.7]]).reshape(1,1)

X = np.array([[1.0,2.8,3.5,2.9],[2.2,3.9,4.8,3.3]])

parameters = {"W1": W1,
              "b1": b1,
              "W2": W2,
              "b2": b2}
#X, parameters = L_model_forward_test_case()

AL, caches = L_model_forward(X, parameters)
print("AL = " + str(AL))
print("Length of caches list = " + str(len(caches)))
print(caches[1])
```

```
AL = [[1. 1. 1. 1.]]
Length of caches list = 2
((array([[ 6.56, 10.57, 12.44,  9.89],
       [12.76, 21.97, 26.34, 20.19],
       [18.06, 34.27, 41.84, 31.49]]), array([[2. , 3.3, 4.3]]), array([[3.7]])), array([[136.586, 244.702, 29
5.414, 225.514]]))
```

Now we have a full forward propagation that takes the input X and outputs a row vector $A^{[L]}$ containing predictions. It also records all intermediate values in "caches". Using $A^{[L]}$, you can compute the cost of your predictions.

5 - Cost function

$$J = -\frac{1}{m} \sum_{i=1}^{m} (y^{(i)} \log\left(a^{[L](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[L](i)}\right))$$

See other version in the summary elsewhere.

In [23]:
```
# This function should be correct as it has been used both by 2-level and L level, and they are all reight.
# Y, AL = compute_cost_test_case()
Y = np.array([[1,0,1,1,0,1,0,0,1,0]]).reshape(10,1)
AL = np.array([[0,1,1,0,1,0,0,1,0,1]]).reshape(10,1)

print(compute_cost(AL, Y))
```

```
[[inf nan nan inf nan inf inf nan inf nan]
 [nan inf inf nan inf nan nan inf nan inf]
 [inf nan nan inf nan inf inf nan inf nan]
 [inf nan nan inf nan inf inf nan inf nan]
 [nan inf inf nan inf nan nan inf nan inf]
 [inf nan nan inf nan inf inf nan inf nan]
 [nan inf inf nan inf nan nan inf nan inf]
 [nan inf inf nan inf nan nan inf nan inf]
 [inf nan nan inf nan inf inf nan inf nan]
 [nan inf inf nan inf nan nan inf nan inf]]

C:\Users\ljyan\Desktop\courseNotes\dataScience\deepLearning\neural network\Part I -- Neural Networks and Deep L
earning\dnn_utils.py:252: RuntimeWarning: divide by zero encountered in log
  cost = (1./m) * (-np.dot(Y,np.log(AL).T) - np.dot(1-Y, np.log(1-AL).T))
```

Need fix the function calculating the cost for in special cases. For example, 0log0 = 0 etc.

6 - Backward propagation module

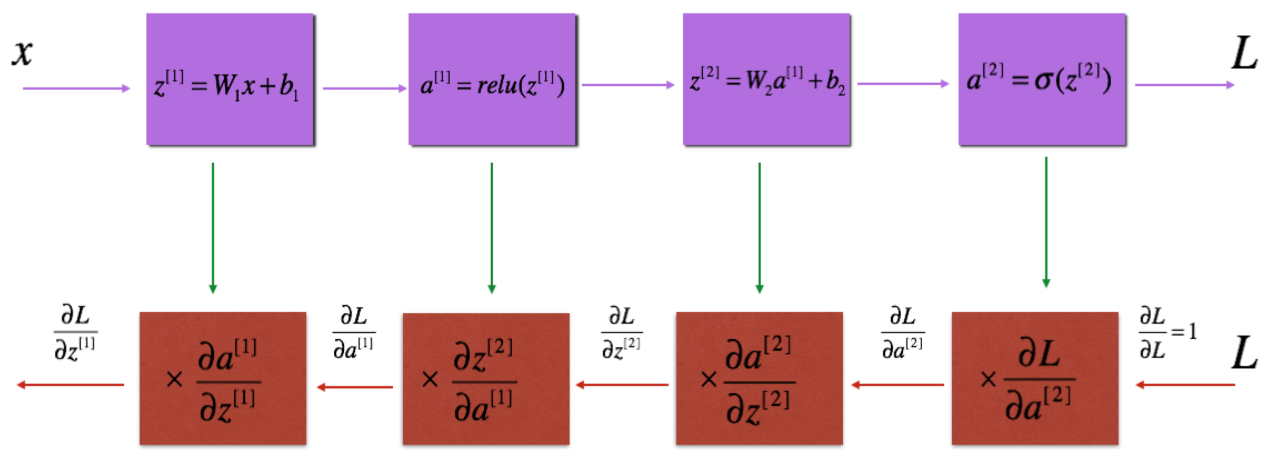

Figure 3 : Forward and Backward propagation for *LINEAR->RELU->LINEAR->SIGMOID*
The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

The chain rule of calculus can be used to derive the derivative of the loss $\mathcal{L}$ with respect to $z^{[1]}$ in a 2-layer network as follows:

$$\frac{d\mathcal{L}(a^{[2]},y)}{dz^{[1]}} = \frac{d\mathcal{L}(a^{[2]},y)}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}}{dz^{[1]}}$$

In order to calculate the gradient $dW^{[1]} = \frac{\partial L}{\partial W^{[1]}}$, use the previous chain rule and obtain $dW^{[1]} = dz^{[1]} \times \frac{\partial z^{[1]}}{\partial W^{[1]}}$. During the backpropagation, at each step you multiply the current gradient by the gradient corresponding to the specific layer to get the gradient you wanted.

Equivalently, in order to calculate the gradient $db^{[1]} = \frac{\partial L}{\partial b^{[1]}}$, use the previous chain rule and obtain $db^{[1]} = dz^{[1]} \times \frac{\partial z^{[1]}}{\partial b^{[1]}}$. This is why we talk about **backpropagation**.

Now, **similar to forward propagation, build the backward propagation in three steps:**

- LINEAR backward
- LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation
- [LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID backward (whole model)

6.1 - Linear backward

For layer l, the linear part is: $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ (followed by an activation).

Suppose you have already calculated the derivative $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$ (This can be calculated with specific activation function as shown later). We want to get $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$.

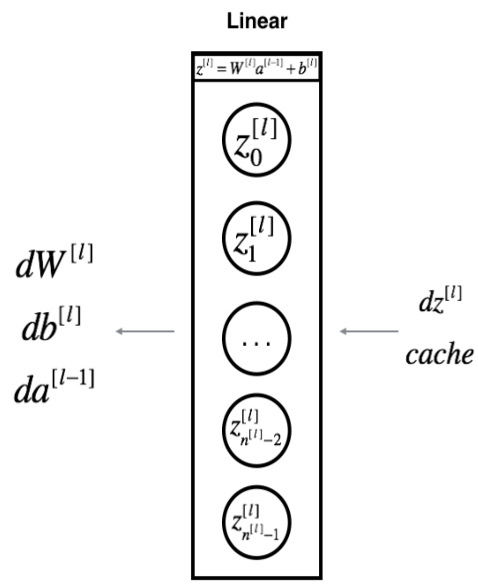

Figure 4

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

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

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

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

In [ ]:
```
# Set up some test inputs
dZ, linear_cache = linear_backward_test_case()

dA_prev, dW, db = linear_backward(dZ, linear_cache)
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

6.2 - Linear-Activation backward

In parallel to the forward propagation, we create a function that merges the two helper functions: **`linear_backward`** and the backward step for the activation **`linear_activation_backward`**.

To help you implement `linear_activation_backward`, we provided two backward functions:

- **`sigmoid_backward`**: Implements the backward propagation for SIGMOID unit. You can call it as follows:

```
dZ = sigmoid_backward(dA, activation_cache)
```

- **`relu_backward`**: Implements the backward propagation for RELU unit. You can call it as follows:

```
dZ = relu_backward(dA, activation_cache)
```

If $g(.)$ is the activation function, `sigmoid_backward` and `relu_backward` compute

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]}) \tag{11}$$

. **Compare this with the two-layer model before**.

Exercise: Implement the backpropagation for the *LINEAR->ACTIVATION* layer.

In [ ]:
```
AL, linear_activation_cache = linear_activation_backward_test_case()

dA_prev, dW, db = linear_activation_backward(AL, linear_activation_cache, activation = "sigmoid")
print ("sigmoid:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db) + "\n")

dA_prev, dW, db = linear_activation_backward(AL, linear_activation_cache, activation = "relu")
print ("relu:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

6.3 - L-Model Backward

Now you will implement the backward function for the whole network. Recall that when you implemented the `L_model_forward` function, at each iteration, you stored a cache which contains (X,W,b, and z). In the back propagation module, you will use those variables to compute the gradients. Therefore, in the `L_model_backward` function, you will iterate through all the hidden layers backward, starting from layer L. On each step, you will use the cached values for layer l to backpropagate through layer l. Figure 5 below shows the backward pass.

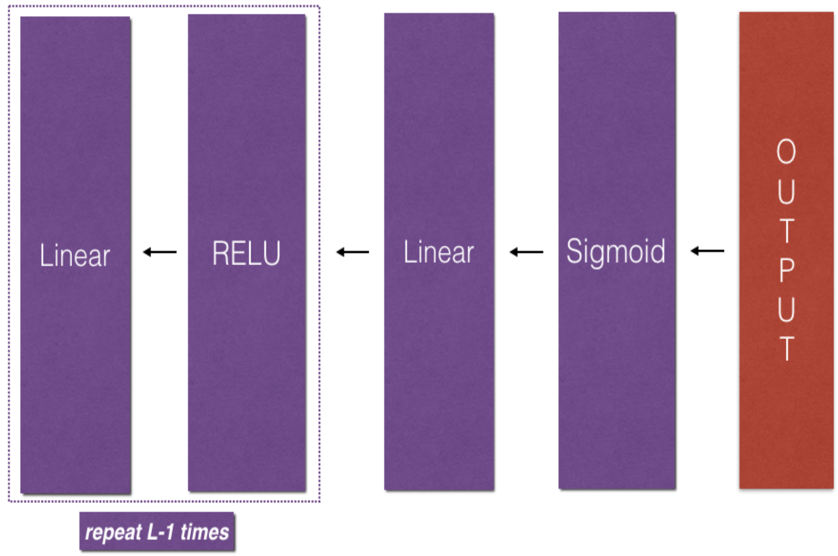

Figure 5 : Backward pass

Initializing backpropagation: To backpropagate through this network, we know that the output is, $A^{[L]} = \sigma(Z^{[L]})$. Your code thus needs to compute `dAL` $= \frac{\partial \mathcal{L}}{\partial A^{[L]}}$. To do so, use this formula (derived using calculus which you don't need in-depth knowledge of):

```
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) # derivative of cost with respect to AL
```

See details in the derivation of a two-layer model in the notes DeepLearningMathematics.

You can then use this post-activation gradient `dAL` to keep going backward. As seen in Figure 5, you can now feed in `dAL` into the LINEAR->SIGMOID backward function you implemented (which will use the cached values stored by the L_model_forward function). After that, you will have to use a `for` loop to iterate through all the other layers using the LINEAR->RELU backward function. You should store each dA, dW, and db in the grads dictionary. To do so, use this formula :

$$grads["\ dW\ " + str(l)] = dW^{[l]} \tag{15}$$

For example, for $l = 3$ this would store $dW^{[l]}$ in `grads["dW3"]` .

Exercise: Implement backpropagation for the *[LINEAR->RELU]* $\times$ *(L-1) -> LINEAR -> SIGMOID* model.

In [ ]:
```
def L_model_backward(AL, Y, caches):

    grads = {}
    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))
    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs: "grads["dAL"], grads["dWL"], gra
    current_cache = caches[-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = linear_backward(sigmoid_backward(dAL,
                                                                                                        current_c

    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 2)], caches". Outputs: "grads["dA" + str(l + 1)] , grads["dW" + str(l + 1
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_backward(sigmoid_backward(dAL, current_cache[1]), current_cache[0
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

In [ ]:
```
X_assess, Y_assess, AL, caches = L_model_backward_test_case()

grads = L_model_backward(AL, Y_assess, caches)
print ("dW1 = "+ str(grads["dW1"]))
print ("db1 = "+ str(grads["db1"]))
print ("dA1 = "+ str(grads["dA1"]))
```

6.4 - Update Parameters

In this section you will update the parameters of the model, using gradient descent:

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]} \tag{16}$$
$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]} \tag{17}$$

where α is the learning rate. After computing the updated parameters, store them in the parameters dictionary.

In [ ]:
```
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
```

In [ ]:
```
parameters, grads = update_parameters_test_case()
parameters = update_parameters(parameters, grads, 0.1)

print ("W1 = " + str(parameters["W1"]))
print ("b1 = " + str(parameters["b1"]))
print ("W2 = " + str(parameters["W2"]))
print ("b2 = " + str(parameters["b2"]))
print ("W3 = " + str(parameters["W3"]))
print ("b3 = " + str(parameters["b3"]))
```