Outline

It is always good to create helper functions to assist you in making a model basically breaking it down into puzzle pieces.

1 Initialize the parameters for a two-layer network and for an L-layer neural network

2. Implement the forward propagation module

- Complete the LINEAR part of a layer's forward propagation step (resulting in Z[l]).
- The ACTIVATION function is provided for you (relu/sigmoid)
- 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.

3. Compute the loss

4. Implement the backward propagation module (denoted in red in the figure below)

- Complete the LINEAR part of a layer's backward propagation step
- The gradient of the ACTIVATION function is provided for you(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

5. Finally, update the parameters

For every forward function, there is a corresponding backward function. This is why at every step of your forward module you will be storing some values in a cache. These cached values are useful for computing gradients.

In the backpropagation module, you can then use the cache to calculate the gradients.

1. Initialize the parameters for a two-layer network and for an L-layer neural network

2 Layer Neural Network function:

```
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)

    """

    np.random.seed(1)

    W1 = np.random.randn(n_h,n_x) * .01 # Small random values

    W2 = np.random.randn(n_y,n_h) * .01

    b1 = np.zeros((n_h,1)) # Np array of zeros

    b2 = np.zeros((n_y,1))

    parameters = {"W1": W1,

               "b1": b1,

               "W2": W2,

               "b2": b2}

 return parameters
```

L Layer Neural Network:

```
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer in our network

    Returns:
    parameters -- python dictionary containing your 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)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims) # number of layers in the network

    for l in range(1, L):
        # So per Layer
        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
```

2. Forward Propagation

- **Complete the LINEAR part of a layer's forward propagation step (resulting in Z[l]).**

```
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, also called pre-activation parameter
    cache -- a python tuple containing "A", "W" and "b" ; stored for computing the backward pass
efficiently
    """
    Z = np.dot(W,A) + b
    cache = (A, W, b) #Storing for backpropagation

    return Z, cache
```

- **The ACTIVATION function is provided for you (relu/sigmoid)**
- **Combine the previous two steps into a new [LINEAR->ACTIVATION] forward function.**

```
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 a text string: "sigmoid" or "relu"

    Returns:
    A -- the output of the activation function, also called the post-activation value
    cache -- a python tuple containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    """

    Z, linear_cache = linear_forward(A_prev,W,b) # Linear part

    if activation == "sigmoid":
        A, activation_cache = sigmoid(Z)
    elif activation == "relu":
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache
```

- **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.**

```
def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computation

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

    Returns:
    AL -- activation value from the output (last) layer
    caches -- list of caches containing:
                every cache of linear_activation_forward() (there are L 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.
    # The for loop starts at 1 because layer 0 is the input
    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)

    return AL, caches
```

3. Compute Cost

```
def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation.
    Arguments:
    AL -- probability vector corresponding to your label predictions, shape (1, number of examples)
    Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat), shape (1, number of examples)
    Returns:
    cost -- cross-entropy cost
    """
    m = Y.shape[1]
    # Compute loss from aL and y.
    cost = -(1/m) * np.sum(Y * np.log(AL) + (1-Y) * np.log(1-AL))
    cost = np.squeeze(cost) # To make sure your cost's shape is what we expect (e.g. this turns [[17]] into 17).

    return cost
```

4. Backward Propagation

- **Complete the LINEAR part of a layer's backward propagation step**

```
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)
    dA_prev = np.dot(W.T, dZ)


    return dA_prev, dW, db
```

- **The gradient of the ACTIVATION function is provided for you(relu_backward/sigmoid_backward)**
- **Combine the previous two steps into a new [LINEAR->ACTIVATION] backward function**

```
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) we store for computing backward
propagation efficiently

    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)


    elif activation == "sigmoid":

        dZ = sigmoid_backward(dA, activation_cache)


    dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

- **Stack [LINEAR->RELU] backward L-1 times and add [LINEAR->SIGMOID] backward in a new L_model_backward function**

```
def L_model_backward(AL, Y, caches):

    """

    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEAR -> SIGMOID group

    Arguments:

    AL -- probability vector, 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 = {}

    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

    # Initial Gradient of Cost with respect to AL
```

```
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    # Compute gradients for the last layer (Sigmoid Activation)
    current_cache = caches[L-1]
    dA_prev, dW, db = linear_activation_backward(dAL, current_cache, activation="sigmoid")

    grads["dA" + str(L-1)] = dA_prev
    grads["dW" + str(L)] = dW
    grads["db" + str(L)] = db

    # Loop through the remaining layers in reverse order (ReLU Activation)
    # Loop from l=L-2 to l=0
    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 1)], current_cache". Outputs: "grads["dA" + str(l)] ,
grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]
        current_cache = caches[l]
        dA_prev, dW, db = linear_activation_backward(grads["dA" + str(l + 1)], current_cache,
activation="relu")

        grads["dA" + str(l)] = dA_prev
        grads["dW" + str(l + 1)] = dW
        grads["db" + str(l + 1)] = db

    return grads
```

5. Update Parameters

```
def update_parameters(params, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    params -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of L_model_backward

    Returns:
    parameters -- python dictionary containing your updated parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """
    parameters = copy.deepcopy(params)
    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)] -= learning_rate * grads["dW" + str(l+1)]
        parameters["b" + str(l+1)] -= learning_rate * grads["db" + str(l+1)]

    return parameters
```

THE MODELS

1.

```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):
    """
    Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 1 if cat, 0 if non-cat), of shape (1, number of examples)
    layers_dims -- dimensions of the layers (n_x, n_h, n_y)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- If set to True, this will print the cost every 100 iterations

    Returns:
    parameters -- a dictionary containing W1, W2, b1, and b2
    """

    np.random.seed(1)
    grads = {}
    costs = []                              # to keep track of the cost
    m = X.shape[1]                          # number of examples
    (n_x, n_h, n_y) = layers_dims
```

```
    # Initialize parameters dictionary, by calling one of the functions you'd previously implemented
    parameters = initialize_parameters(n_x, n_h, n_y)

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1, W2, b2". Output: "A1, cache1, A2, cache2".
        A1, cache1 = linear_activation_forward(X, W1, b1, activation="relu")
        A2, cache2 = linear_activation_forward(A1, W2, b2, activation="sigmoid")

        # Compute cost
        cost = compute_cost(A2, Y)

        # Initializing backward propagation
        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; also dA0 (not used), dW1, db1".
```

```
    dA1, dW2, db2 = linear_activation_backward(dA2, cache2, activation="sigmoid")
    dA0, dW1, db1 = linear_activation_backward(dA1, cache1, activation="relu")

    # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db2'] to db2
    grads['dW1'] = dW1
    grads['db1'] = db1
    grads['dW2'] = dW2
    grads['db2'] = db2

    # Update parameters.
    parameters = update_parameters(parameters, grads, learning_rate)

    # Retrieve W1, b1, W2, b2 from parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Print the cost every 100 iterations
    if print_cost and i % 100 == 0 or i == num_iterations - 1:
        print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
    if i % 100 == 0 or i == num_iterations:
        costs.append(cost)

return parameters, costs
```

```
def plot_costs(costs, learning_rate=0.0075):

    plt.plot(np.squeeze(costs))

    plt.ylabel('cost')

    plt.xlabel('iterations (per hundreds)')

    plt.title("Learning rate =" + str(learning_rate))

    plt.show()
```

2.

```
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):

    """

    Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.


    Arguments:

    X -- input data, of shape (n_x, number of examples)

    Y -- true "label" vector (containing 1 if cat, 0 if non-cat), of shape (1, number of examples)

    layers_dims -- list containing the input size and each layer size, of length (number of layers + 1).

    learning_rate -- learning rate of the gradient descent update rule

    num_iterations -- number of iterations of the optimization loop

    print_cost -- if True, it prints the cost every 100 steps


    Returns:

    parameters -- parameters learnt by the model. They can then be used to predict.
```

```
"""

np.random.seed(1)
costs = []                         # keep track of cost

# Parameters initialization.
parameters = initialize_parameters_deep(layers_dims)

# Loop (gradient descent)
for i in range(0, num_iterations):

    # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
    AL, caches = L_model_forward(X, parameters)

    # Compute cost.
    cost = compute_cost(AL, Y)

    # Backward propagation.
    grads = L_model_backward(AL, Y, caches)

    # Update parameters.
    parameters = update_parameters(parameters, grads, learning_rate)

    # YOUR CODE ENDS HERE
```

```
    # Print the cost every 100 iterations
    if print_cost and i % 100 == 0 or i == num_iterations - 1:
        print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
    if i % 100 == 0 or i == num_iterations:
        costs.append(cost)

return parameters, costs
```